Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Fakultät für Mathematik und Informatik
Institut für Informatik
Mensch-Computer-Interaktion

Masterarbeit

zur Erlangung des Grades
Master of Science (M.Sc.)
im Studiengang Informatik
an der Universität Würzburg

Visualisierung von Software-Historie in Virtual Reality

Erweiterung einer Insel-Metapher zur Darstellung der Evolution OSGi-basierter Software-Projekte

vorgelegt von
Elke Franziska Heidmann
Matrikelnummer: 1982043
heidmann.elke@t-online.de

am 9. November 2020

Betreuer/Prüfer:
Prof. Marc Erich Latoschik, Informatik IX, Universität Würzburg

Betreuung am
Deutsches Zentrum für Luft- und Raumfahrt
Institut für Simulations- und Softwaretechnik
Lynn von Kurnatowski
Andreas Schreiber

Zusammenfassung

Software-Visualisierungen mit Metaphern bieten einen einfachen Zugang zum Verständnis der Systemarchitektur. Die Darstellung durch eine Inselmetapher eignet sich dabei besonders für modulare Software-Projekte.
Neben dem aktuellen Stand des Projekts ist auch dessen Historie von Bedeutung. Aus dieser lassen sich unter anderem logische Abhängigkeiten ableiten, die die Planung der weiteren Entwicklung und Wartung der Software unterstützen können.
Folglich ist auch eine Visualisierung der Software-Historie sinnvoll. Momentan gibt es keine Darstellung, die eine Inselmetapher nutzt und gleichzeitig die zeitliche Komponente der Software-Entwicklung abbilden kann.

Diese Arbeit entwickelt daher ein Konzept zur Erweiterung der Inselmetapher um eine Historiendarstellung von Software-Projekten. Der Schwerpunkt ist dabei, die mentale Karte des Nutzers zu erhalten, während sich die Bestandteile der Inselwelt aufgrund der Weiterentwicklung der dargestellten Software verändern.
Das Konzept wird prototypisch in die Anwendung IslandViz integriert, die bereits durch eine Inselmetapher Software-Architekturen statisch in einer virtuellen Umgebung darstellt.

Die anschließende Nutzerstudie vergleicht den entwickelten Prototyp mit einer bereits existierend Visualisierung, die Software-Architekturen zu verschiedenen Zeitpunkten unabhängig voneinander darstellt. Dabei zeigt sich, dass es für die Identifizierung von Veränderungen hilfreich ist, dass der vom Nutzer gewonnene Überblick in der Visualisierung im zeitlichen Verlauf erhalten bleibt. Nutzer lösen Aufgaben im Prototyp durchschnittlich bis zu 50% schneller und geben 33% mehr richtige Antworten auf Fragen zu dargestellten Änderungen. Die wahrgenommene geistige Anstrengung ist dabei um 21% geringer. Weiterhin werden durch die Studie Ansätze zur weiteren Verbesserung des Konzepts identifiziert.

Das hier entwickelte Konzept zur Visualisierung der Historie modularer Software-Projekte durch die Inselmetapher wird zukünftig die Planung von Erweiterungs- und Wartungsaufgaben unterstützen.

Abstract

Software visualisations using metaphors can provide an intuitive approach to the system's architecture. An island metaphor is especially suitable for modular software systems.
Apart from the current status of a project, its history is a crucial factor. It provides, among others, information about logical dependencies to help plan tasks for further development and maintenance of the software.
Therefore, a visualisation of software history would be useful. At the moment, there are no visualisations using an island metaphor existing which simultaneously represent the history of the software project.

This thesis presents a concept to extend the island metaphor in order to visualise software history. Its focus is to maintain the user's mental map of the representation while the components of the archipelago are continuously adapted due to changes of the software system.
The concept is implemented as a prototype integrated in the application IslandViz . This application already shows static software architecture in a virtual environment using an island metaphor.

A subsequent user study compares the developed prototype to an existing, static software visualisation, which depicts software architectures independently at different points in time. The study shows that in order to identify changes, it is helpful that the user's overview of the visualisation is kept over time. Users, on average, solve tasks up to 50% faster and answer questions regarding the depicted changes 33% more accurately in the newly developed prototype. At the same time, the user's perceived mental effort is about 21% lower. In addition, the study reveals aspects for further improvement of this concept.

The presented concept to visualise the history of modular software projects using an island metaphor will help plan development and maintenance tasks in the future.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Akronyme

ASQ After-Scenario Questionnaire. 71, 75, 76, 82

DLR Deutsches Zentrum für Luft- und Raumfahrt. 3

EHTA Enhanced Hexagon Tiling Algorithm. 33, 34, 36, 60

HMD Head-Mounted Display. 71, 75

NASA-TLX NASA Task Load Index. 71, 75, 76, 84, 94

OSGi Open Services Gateway Initiative. 6, 7, 11, 12, 18

RCE Remote Computing Environment. 91

SSQ Simulation Sickness Questionnaire. 71, 74–76, 86

SUS System Usability Scale. 71, 75, 76, 83

UML Unified Modeling Language. 1

VR Virtual Reality. 1, 2, 8, 46, 54, 65, 75

1 Einleitung

1.1 Motivation

Lehman (1996) postuliert, dass Software-Produkte im Laufe ihres Lebenszyklus kontinuierlich angepasst werden müssen, da sich die Umgebung, in der sie eingesetzt werden, mit der Zeit verändert. Zusätzlich steigt der Umfang der Software, da neben Anpassungen vom Nutzer auch Erweiterungen der Funktionalität gewünscht werden. Weitere Gründe für Veränderungen am Software-Projekt sind nach ISO/IEC-14764 (2006) das Korrigieren von Fehlern und die Verbesserung der Software, z. B. im Hinblick auf Performance und Wartbarkeit.
Der Begriff Software-Historie beschreibt die Veränderungen und Erweiterungen, die im Laufe der Entwicklungs- und Maintenance-Phasen eines Software-Projekts gemacht werden.
Durch die Anpassungen nimmt die Komplexität der Software zu, da sukzessive zusätzliche, unstrukturierte Abhängigkeiten hinzukommen. Auch gezielte Gegenmaßnahmen können diesen Effekt nicht vollständig aufheben. Gleichzeitig steigt die Herausforderung, dass alle an der Software-Entwicklung Beteiligten trotz kontinuierlicher Änderungen den Überblick über das System behalten (Lehman, 1996).

Als Zugang zu Software-Projekten und ihrer Architektur werden üblicherweise graphische Darstellungen verwendet, bei denen die Entitäten der Software durch einfache, geometrische Formen repräsentiert werden. Hierfür ist das UML-Diagramm das prominenteste Beispiel.
Verschiedene Arbeiten beschäftigen sich damit, alternative Visualisierungen für Software-Architekturen zu entwickeln. Dabei werden komplexere graphische Elemente, dreidimensionale Visualisierungen, die Verwendung von Metaphern oder Darstellungen in Virtual Reality (VR) untersucht. Die Aspekte können außerdem kombiniert werden. Die Verwendung der dritten Dimension in Visualisierungen bietet die Möglichkeit, eine höhere Informationsdichte zu erzielen. So nutzen Marcus, Feng und Maletic (2003) diese, um zusätzliche Software-Metriken abzubilden. Werden

Software-Systeme durch Metaphern dargestellt, bietet sich dem Betrachter durch die aus dem Alltag bekannten Objekte ein intuitiverer Zugang. So verwenden Panas, Berrigan und Grundy (2003) und Wettel und Lanza (2007) jeweils eine Stadtmetapher, um Software-Systeme zu visualisieren. Die Darstellung als Inselwelt, wie bei Misiak (2017), stellt die modulare Architektur der visualisierten Software in den Vordergrund. Durch die Verwendung von VR und die damit verbundene Immersion wird eine stärkere Konzentration des Nutzers auf die Visualisierung erreicht. Diese Art der Visualisierung steigert außerdem die Zufriedenheit und das Erinnerungsvermögen der Nutzer (Merino et al., 2017).

Software-Produktanager und Entwickler müssen im Entwicklungsprozess Entscheidungen zur weiteren Gestaltung der Software treffen. Zusätzlich zum Verständnis der aktuellen Software-Architektur können die Ergebnisse einer Analyse der im Software-Projekt anfallenden Daten dabei helfen, diesen Prozess zu verbessern. Die von Menzies und Zimmermann (2013) anhand dieses Nutzens definierte Software-Analyse untersucht unter anderem logische Abhängigkeiten (Gall, Hajek & Jazayeri, 1998) im Software-Projekt auf Basis von dessen Historie. Diese beschreiben Beziehungen "zwischen Quellcode-Dateien, die oft zusammen geändert werden, obwohl zwischen ihnen nicht notwendigerweise eine strukturelle Abhängigkeit besteht". (Oliva, Santana, Gerosa & Souza, 2011).

Die Kenntnis über logische Abhängigkeiten im Software-Projekt kann dessen weitere Entwicklung im Hinblick auf zwei Aspekte verbessern: Zum einen ist es laut Mockus und Weiss (2000) wahrscheinlicher, dass Fehler auftreten, wenn sich eine Änderung über mehrere Dateien, Module oder Subsysteme erstreckt. Cataldo, Mockus, Roberts und Herbsleb (2009) fanden zudem heraus, dass der Einfluss logischer Abhängigkeiten auf auftretende Fehler größer ist als der Einfluss syntaktischer Abhängigkeiten. Zum anderen lässt sich anhand logischer Abhängigkeiten der Umfang zukünftiger Änderungen abschätzen: Gall et al. (1998) schlagen vor, die Release-Historie einer Software zu untersuchen, um Muster zu finden, bei denen z. B. Module immer gemeinsam geändert werden. Analog dazu identifizieren Oliva et al. (2011) als einen Grund für logische Abhängigkeiten, dass eine Gruppe von Artefakten überarbeitet wird, die gemeinsam einer Funktionalität oder einer Rolle innerhalb der Architektur zugeordnet sind. Durch ein einfaches Identifizieren dieser semantischen Klassen können die Anpassbarkeit der Software verbessert und die Planung von Maintenance-Aufgaben unterstützt werden, da die betroffenen Artefakte vorher bekannt sind.

1.2 Problembeschreibung

Da somit in der Software-Historie relevante Informationen zu finden sind, erscheint es hilfreich, auch den zeitlichen Aspekt eines Software-Projekts visualisieren zu können (Beyer und Hassan (2006)). Da einfache, geometrische Darstellungen großer Software-Projekte bei Berücksichtigung der zeitlichen Komponente schnell unübersichtlich werden können, wird diese wiederum in Visualisierungen integriert, die Metaphern verwenden. So ist mit EvoStreets (Steinbrückner & Lewerentz, 2010) eine Stadtmetapher verwendet worden, um Software-Historie darzustellen.
Eine Inselmetapher, die sich besonders zur Darstellung modularer Software eignet, ist bisher nicht für die Darstellung von Software-Historie verwendet worden.

1.3 Beitrag der Arbeit

Im Rahmen dieser Arbeit soll ein Konzept erstellt werden, wie die Visualisierung von Software-Architektur durch eine Inselmatapher erweitert werden kann, um zusätzlich die Software-Historie darzustellen.

Verwandte Arbeiten beschreiben den Erhalt einer mentalen Karte (Eades, Lai, Misue & Sugiyama, 1991) als wichtige Voraussetzung zur Visualisierung von Software-Historie. Dabei sollen die Repräsentationen von Software-Artefakten im Laufe der dargestellten Zeit stets an ähnlichen Positionen bleiben, damit der Nutzer seinen bereits erarbeiteten Überblick im nächsten betrachteten Zeitpunkt weiter verwenden kann und sich in der Visualisierung nicht neu orientieren muss.
Ein Schwerpunkt dieser Arbeit ist, den Erhalt der mentalen Karte in das zu entwickelnden Konzept zu einzubringen.
Weiterhin wird eine nutzerzentrierte Anforderungsanalyse in Form eines Fokusgruppengesprächs durchgeführt. Diese Analyse beleuchtet vor allem, welche Informationen Software-Entwickler aus der Software-Historie verwenden. Hieraus sollen weitere funktionale Anforderungen abgeleitet werden, die in das Konzept integriert werden.

Das Konzept zur Visualisierung von Software-Historie durch eine Inselmetapher wird prototypisch in die Anwendung IslandViz (Misiak, 2017) des DLR -Instituts für

Simulations- und Softwaretechnik integriert. Diese Anwendung verwendet eine Inselmetapher um OSGi-basierte Software-Architekturen statisch zu visualisieren. Die Anwendung unterstützt bisher nicht die Darstellung von Software-Historie und eignet sich daher, das entwickelte Konzept zu demonstrieren.

Durch eine Nutzerstudie an dem erstellten Protoyp wird schließlich das Konzept evaluiert. Dabei liegt der Schwerpunkt einerseits darauf zu zeigen, dass der Erhalt der mentalen Karte Vorteile zur Identifizierung von Änderungen am dargestellten Software-Projekt bietet. Andererseits soll die Nutzerinteraktion mit der Software-Historie bewertet und Potential für Verbesserungen gefunden werden.

Das im Rahmen dieser Arbeit erstellte Konzept bietet die Möglichkeit, Software-Historie und damit die zeitlichen Änderungen des Projekts durch eine Inselmetapher zu visualisieren. In einem weiteren Schritt könnten auch daraus abgeleitete Analyse-Ergebnisse wie logische Abhängigkeiten dargestellt werden.
Für zukünfige Arbeiten bietet sich außerdem die Möglichkeit, zu untersuchen, ob sich die Visualisierung umgekehrt auch eignet, logische Abhängigkeiten mit Hilfe der menschlichen Fähigkeit zur Mustererkennung zu identifizieren. Die so erkannten semantischen Klassen könnten ausreichend sein, die Planung von Maintanance-Aufgaben zu unterstützen.

1.4 Gliederung der Arbeit

Die vorliegende Arbeit ist in folgende Abschnitte gegliedert: Zunächst werden die für die Arbeit grundlegenden Begriffe definiert, und verwandte Arbeiten aus der Software-Visualisierung vorgestellt. In Kapitel 3 werden die Anforderungen an das Konzept zur Erweiterung erarbeitet und die zukünftige Verwendung der Visualisierung wird erläutert. Die Konzeption der Visualisierung von Software-Historie mittels einer Inselmetapher erfolgt in Kapitel 4. Es folgen Details zur Implementierung. Die Nutzerstudie zur Evaluation wird in Kapitel 6 erläutert und ausgewertet. Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick ab.

Die Arbeit verwendet zur besseren Lesbarkeit das generische Maskulinum. Damit werden jedoch stets weibliche und männliche Nutzer, Teilnehmer, Probanden usw. beschrieben.

2 Grundlagen & Verwandte Arbeiten

In diesem Kapitel werden die für die Arbeit grundlegenden Begriffe Software-Architektur, Software-Visualisierung erläutert und die zum Verständnis der Konzeption notwendigen Details dargelegt.

2.1 Software-Architektur

Der Begriff der Software-Architektur wird in der Praxis unterschiedlich definiert und verwendet. Nach ISO/IEC/IEEE-42010 (2011) wird die Architektur eines Software-Systems als seine grundlegenden Konzepte und Eigenschaften definiert:

> *"fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution"*

Dabei kann die Software-Architektur in verschiedene Aspekte untergliedert werden (Hofmeister, Nord & Soni, 1999):

- **conceptual view:** Hierbei werden die Funktionalitäten beschrieben und das System nach diesem Aspekt in Komponenten gegliedert. Zusätzlich werden die Schnittstellen beschrieben, durch die eine Interaktion mit dem System stattfinden kann. Hierbei ist darauf zu achten, dass Funktionalitäten geändert, hinzugefügt, entfernt oder ausgetauscht werden können.
- **module view:** Bei dieser Betrachtung werden die einzelnen Subsysteme in Module aufgeteilt, die Layers zugeordnet werden. Diese Layer sind jeweils bestimmten Sorten von Hardware oder externen Komponenten zugeordnet.
- **execution view:** Die Ausführung des Systems wird mit diesem Aspekt beschrieben. Dies umfasst die Zuweisung der Module zu physikalischen Resourcen, sowie die Kommunikation zwischen den Modulen.

- **code view:** Die Zuordnung von Modulen und Interfaces, die im module view definiert werden, zu Sourcecode-Dateien und ihre Organisation in Ordnern wird durch diese Sicht beschrieben.

2.1.1 OSGi

Bei der Open Services Gateway Initiative (OSGi) handelt es sich um eine Plattform, die die Implementierung komponentenbasierter und serviceorientierter Software-Systeme ermöglicht (Tavares & Valente, 2008), die in Java geschrieben werden. OSGi legt somit die Grundlagen der Architektur des Software-Projekts fest.

Bei klassischen Java-Systemen werden die einzelnen Quellcode-Dateien (.jar) in Packages verwaltet. Diese Pakete können als Hierarchie geschachtelt werden und bieten durch Namespaces eine Möglichkeit zur Modularisierung. Die einzelnen Dateien dienen als Verarbeitungseinheiten für den Compiler, weshalb sie auch als CompilationUnits bezeichnet werden. Jeder CompilationUnit ist eine Klasse zugeordnet, die auch weitere, verschachtelte Klassen beinhalten kann. Für die Modularisierung großer Software-Projekte ist diese Strukturierung jedoch nicht ausreichend. Zudem kann keine ausreichende Kapselung vorgenommen werden, da öffentliche Klassen im gesamten System sichtbar sind.
Dieses Problem wird durch das OSGi-Framework gelöst, das die "Grundlage zur Erstellung und Ausführung lose gekoppelter, dynamisch modularer Systeme" (McAffer, VanderLei & Archer, 2010) bildet: Die Modularisierung wird durch Bundles ermöglicht. Dabei handelt es sich um eine abgeschlossene Menge von in Packages organisierten CompilationUnits. Gemäß der gängigen Definition von Modularisierung schirmen Bundles ihre interne Implementierung nach außen ab und definieren ihre Abhängigkeiten und Schnittstellen von und zu anderen Bundles durch Import- und Export-Anweisungen. Durch den Export eines Packages macht das Bundle dessen Inhalt für andere Bundles sichtbar und nutzbar. Um den exportierten Inhalt eines anderen Bundles zu nutzen, muss dieser bei Bedarf explizit importiert werden.

Durch die OSGi-Plattform können Bundles zusätzlich zu ihren Import-Abhängigkeiten durch ein Service-Layer interagieren. Durch dieses Konzept wird eine stärkere Entkopplung erreicht. Services werden durch klassische Java-Interfaces definiert, deren CompilationUnits öffentlich und in exportierten Packages abgelegt sein müssen. In beliebigen Bundles kann der Service mit Funktionalitäten implementiert werden.

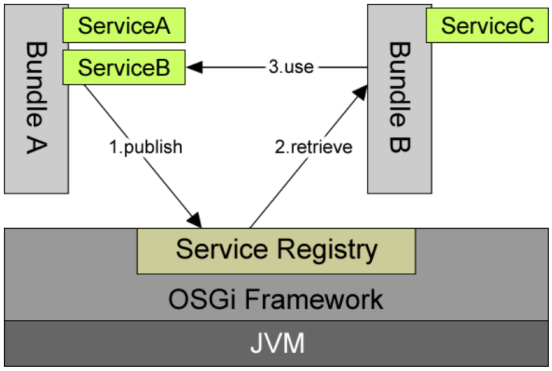


Abbildung 2.1. Visualisierung des Konzepts der OSGi-Services (Tavares & Valente, 2008)

Das zugehörige Bundle instanziiert den Service und registriert ihn im Service Registry der Anwendung. Die CompilationUnits und Packages, die für die Implementierung zuständig sind, sind dabei im Allgemeinen nicht öffentlich und werden nicht exportiert. Hierdurch entsteht die Entkopplung. Um den Service zu nutzen, muss ein Bundle das Package der Service-Definition importieren und sucht im Service Registry nach einer passenden Instanz, um diese zu nutzen. Dies wird in Abbildung 2.1 dargestellt.(McAffer et al., 2010), (Tavares & Valente, 2008)

Das OSGi-Framework gibt zusammengefasst eine Richtlinie zur Gestaltung der Software-Architektur auf module-, execution- und code-Ebene vor: Durch die Bundles wird das System modularisiert. Der Service-Layer organisiert die Kommunikation zwischen Modulen. Die Interfaces und Funktionalitäten der Module werden in CompilationUnits und durch Package-Hierarchien strukturiert.

2.2 Software-Visualisierung

2.2.1 Grundlagen der Software-Visualisierung

Bei Software-Visualisierung handelt es sich nach Diehl (2007) um grafische Repräsentationen der Struktur, des Verhaltens oder der Entwicklung von Software. Bei Strukturen handelt es sich um die statischen Aspekte der Software, die identifiziert werden können, ohne das die Software ausgeführt werden muss. Dazu gehören die Datenstrukturen und die Organisation des Programms in Module. Dieser Aspekt wird in Form der Software-Architektur in dieser Arbeit betrachtet. Bei der Entwicklung der Software handelt es sich um die bereits in der Einleitung beschriebene Software-Historie. Das Verhalten einer Software während ihrer Ausführung wird nicht weiter betrachtet.
Diehl (2007) definiert das Ziel von Software-Visualisierung damit, das Verständnis der Software zu unterstützen und die Produktivität des Entwicklungsprozesses zu steigern.

Metaphern

Bei Metaphern in der Visualisierung handelt es sich um eine graphische Repräsentation durch die ein abstraktes Objekt oder Konzept dargestellt wird. Dabei werden domainspezifische Eigenschaften der abstrakten Entität in Eigenschaften der Domaine der Repräsentation übertragen (Diehl (2007)). Der Vorteil der Verwendung von Metaphern in der Visualisierung ist ein intuitiveres Verständnis der Darstellung.

Wahl des Darstellungsmedium

Zur Software-Visualisierung bieten sich unterschiedliche Median an. Am häufigsten wird eine Darstellung am Bildschirm oder auf Papier gewählt. Der Prototyp dieser Arbeit wird hingegen als Virtual Reality (VR) Anwendung realisiert.

Merino et al. (2017) untersuchten den Einfluss des Mediums der Software-Visualisierung auf Leistung (Zeit zum bearbeiten der Aufgbabe und Richtigkeit), Erinnerungsvermögen an die Darstellung und User-Experience (Emotionen und empfundenen Schwierigkeit) von Software-Entwicklern. Dazu wurden neun Einfache Fragen gestellt, die an Visualisierungen von Software-Systemen durch eine Stadtmetapher (Wettel & Lanza, 2007) beantwortet werden sollten. Dabei wurde die Visualisierung

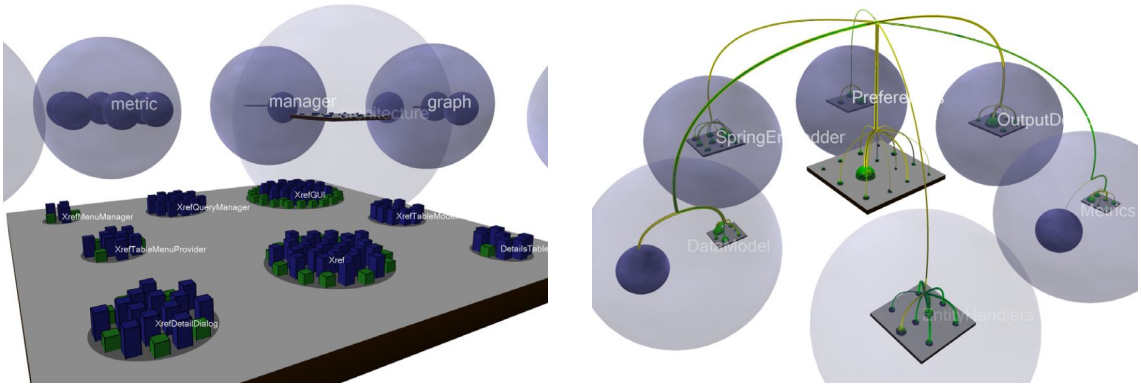
auf drei Medien, einem Computer-Bildschirm, einer immersiven 3D Umgebung und als 3d gedrucktes Model dargestellt. Merino et al. (2017) fanden unter anderem heraus, dass die verwendung einer VR-Visualisierung zu einer höheren Motivation der Probanden führte. Die Aufgaben wurden zwar als schwieriger wahrgenommen, die Probanden äußerten sich jedoch positiver. Bei der Navigation und Orientierung wurde die zusätzliche Dimension als hilfreich empfunden. Die Probanden konnten sich im Anschluss an das Experiment am detailliertesten an das System erinnern.

2.2.2 Verwandte Arbeiten

Im folgenden Abschnitt wird eine Auswahl verschiedener Arbeiten vorgestellt, in denen Software-Architektur und/oder Software-Historie visualisiert werden. Im Bereich der Darstellung von Software-Architektur ohne zeitliche Komponente gibt es viele Beispiele für die Verwendung von dreidimensionalen Darstellungen und Metaphern. Da IslandViz und die in der vorliegenden Arbeit erstellte Erweiterung ebenfalls dieses Visualisierungskonzept verwenden, werden dazu passende Arbeiten vorgestellt. Im Bereich der Visualisierung von Software-Historie werden diese Aspekte hingegen bisher selten verwendet.
Bei den meisten vorgestellten Visualisierungen handelt es sich um akademische Prototypen.

Software-Architektur als Städte und Landschaften

Die Software-Architekturen und die Strukturierung von Quelldateien in Hierarchien können Mithilfe sogenannter Landschaftsmetaphern unterschiedlicher Komplexität dargestellt werden.
Balzer, Noack, Deussen und Lewerentz (2004) nutzen eine “Software Landscape”, um Software-Systeme darzustellen, die sich aus Paketen, Klassen, Methoden und Attributen zusammensetzten. Dabei wird die beliebig tiefe Paket-Hierarchie durch ineinander geschachtelte Kugeln dargestellt. Innerhalb jeder Kugel befindet sich eine Plattform, auf der die im Paket enthaltenen Klassen als Kreise dargestellt werden. In diesen befinden sich Quader und Würfel, die die zugehörigen Methoden und Attribute repräsentieren (Abbildung 2.2a).

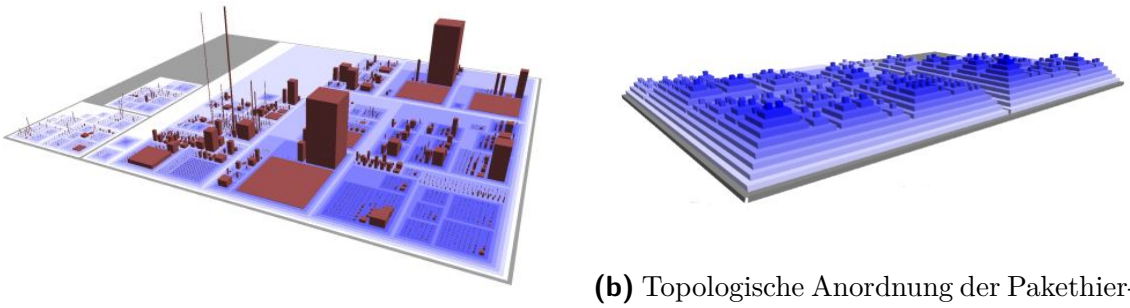


(a) Strukturansicht **(b)** Beziehungen

Abbildung 2.2. Software Landschaft: Paket-Hierarchien durch geschachtelte Kugeln, enthaltende Klassen mit Methoden und Attributen auf Plattform, Beziehungen als gebündelte Verbindungen (Balzer, Noack, Deussen & Lewerentz, 2004)

Bei der räumlichen Navigation werden je nach Standort entfernt oder in der Hierarchie untergeordnet liegende Kugeln undurchsichtig dargestellt, um ihren Inhalt zu verbergen und die Komplexität gering zu halten.
Die Visualisierung ermöglicht außerdem die Darstellung von Beziehungen durch Vererbung, Methoden-Aufruf und Attribut-Zugriff. Die entsprechenden Verbindungen werden zur besseren Übersichtlichkeit in einem “Hierarchical Net” Balzer et al. (2004) über die Packages gebündelt (Abbildung 2.2b). Der Nutzer kann zur besseren Übersicht gezielt Verbindungen eines Typs oder zwischen bestimmten Objekten ein- und ausblenden.

Paket-Hierarchien und Klassen werden ebenfalls durch die City-Metapher von Wettel und Lanza (2007) dargestellt. Dabei werden rechteckige, geschachtelte Bezirke zur Darstellung von Paketen genutzt, die zur besseren Abgrenzung in verschiedenen Farbintensitäten dargestellt werden. In den Bezirken werden die Klassen als Gebäude platziert, deren Maße sich nach grundlegenden Metriken richten: die Höhe wird durch die Anzahl der Methoden, die Grundfläche durch die Anzahl der Attribute bestimmt (Abbildung 2.3a).
Zusätzlich schlagen Wettel und Lanza (2007) vor, eine Topologie einzuführen, bei der die Pakete als aufeinander gestapelte Plattformen betrachtet werden. Hierdurch werden die Bezirke in verschiedenen Höhen entsprechend ihrer Tiefe in der Hierarchie dargestellt (Abbildung 2.3b).
Die Stadt-Metapher ermöglicht ebenfalls eine Navigation durch die Darstellung und das Filtern von Elementen.



(a) Systemüberblick

(b) Topologische Anordnung der Pakethierarchien

Abbildung 2.3. Stadt-Metapher: Paket-Hierarchie als geschachtelte Bezirke, Klassen als Gebäude mit Metrik-basierten Maßen (Wettel & Lanza, 2007)

Panas et al. (2003) verwenden ebenfalls eine Stadtmetapher, die im Gegensatz zu Wettel und Lanza (2007) mehr Detailreichtum aufweist. In diesem Fall werden Pakete als Städte und Klassen als Gebäude dargestellt, deren Höhe wiederum durch Metriken festgelegt ist. Die Dichte der Gebäude soll zusätzlich die Kopplung zwischen den Klassen widerspiegeln. Das gesamte Software-System wird als eine Landschaft visualisiert, wobei Beziehungen zwischen Paketen durch verbindende Wasserwege und Straßen realisiert sind. Um die Metapher einer Stadt eingängiger zu machen, enthält die Visualisierung zusätzliche, nicht funktionale Elemente, wie Straßenlaternen und Bäume.
Im Gegensatz zu den anderen vorgestellten Arbeiten veranschaulicht diese Visualisierung auch dynamische Eigenschaften der Software, indem sich Fahrzeuge zwischen Gebäuden und Städten bewegen, wobei deren Geschwindigkeit und Art beispielsweise Performance-Aspekte und Prioritäten abbilden.
Zusätzlich zu dynamischen und statischen Informationen der Software bieten Panas et al. (2003) Ansichten an, die Informationen über Zuständigkeiten und den Entwicklungsstand darstellen (Abbildung 2.4). So können Komponenten, an denen momentan gearbeitet wird, farblich hervorgehoben und der zuständige Entwickler angegeben werden. Passende Metaphern zeigen Komponenten an, die obsolet sind (braun), häufig ausgeführt werden (Flammen), häufig geändert werden (Blitze) oder seit längerem nicht überarbeitet wurden (verfallenes Gebäude).

Misiak (2017) verwendet in der Anwendung IslandViz ebenfalls eine Landschaftsmetapher. Konkret wird eine Darstellung als Inselwelt gewählt. IslandViz ist auf die Darstellung modularer, OSGi-basierter Software-Architektur spezialisiert. Da diese in Bundles, Packages und CompilationUnits geglieder ist, bietet sich die Inselmetapher an, um die verschiedenen Hierarchieebenen deutlich voneinander abzugrenzen.

Abbildung 2.4. Stadtmetapher mit Projektinformationen (Panas, Berrigan & Grundy, 2003)

Die einzelnen Bundles werden durch Inseln repräsentiert. Ihre Fläche ist in Regionen aufgeteilt, die den im Bundle enthaltenen Packages entsprechen. Die einzelnen CompilationUnits (Klassen und Interfaces) sind verschiedenfarbige Gebäude, die sich in den entsprechenden Regionen befinden (Abbildung 2.5 a-c).
Passend zur Darstellung als Inselwelt werden die import-bedingten Abhängigkeiten zwischen Bundles durch Pfeile dargestellt, die zwischen den Häfen der Inseln (Abbildung 2.5 d) verlaufen.
Das OSGi-spezifische Service-Layer wird durch ein weiteres Liniennetz über den Inseln dargestellt.

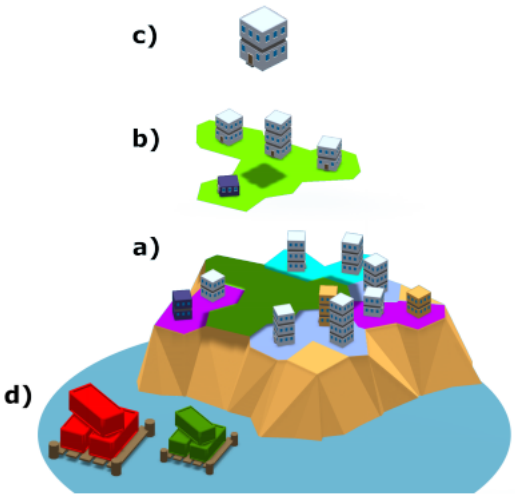
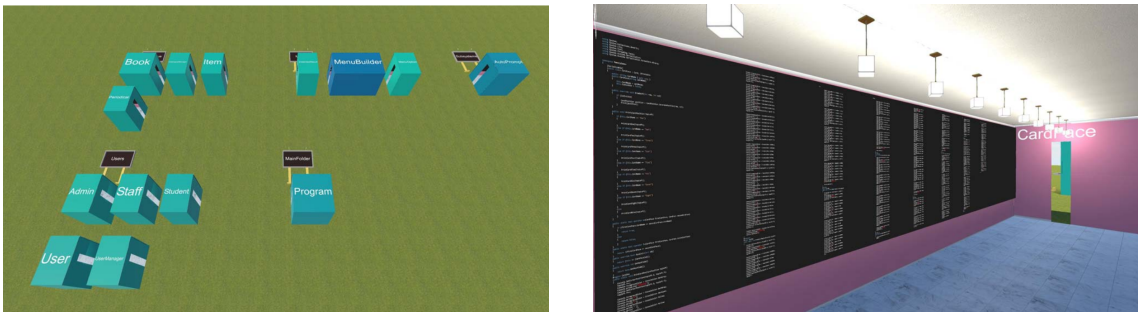


Abbildung 2.5. Darstellung der Elemente von OSGi-Architekturen als Bestandteile einer Inselmetapher (Schreiber, Nafeie, Baranowski, Seipel & Misiak, 2019)



(a) Systemübersicht **(b)** Innenraum mit Quellcode an der Wand

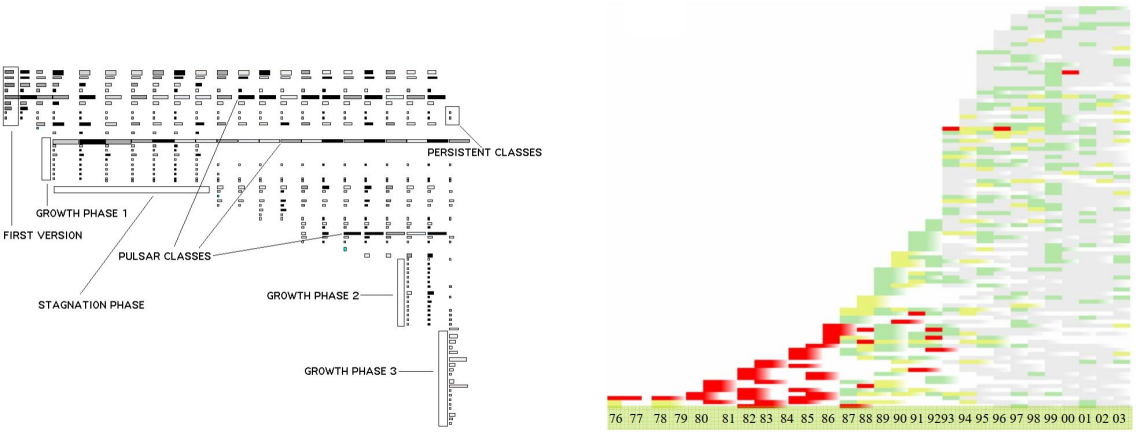
Abbildung 2.6. Code Park: Landschaftsmetapher mit Schnittstelle zum Source Code (Khaloo, Maghoumi, Taranta, Bettner & Laviola, 2017)

Visualisierungen mit Fortgeschrittenen Funktionalitäten

Weitere Arbeiten ermöglichen neben der Darstellung der Software-Architektur mittels Metaphern auch Zugriff zu spezialisierten Funktionalitäten. Mit Code Park visualisieren Khaloo, Maghoumi, Taranta, Bettner und Laviola (2017) die Organisation der Klassen eines Software-Projekts als Gebäude auf einer Rasenfläche (Abbildung 2.6a). Hierbei wird ein ähnlicher Ansatz wie bei Wettel und Lanza (2007) gewählt, der die Größe der Klasse auf die der Gebäude abbildet. Um eine Schnittstelle zum Quellcode zu bieten, werden an den Wänden im Innenraum der Gebäude die Attribute der Klasse und der Code angezeigt (Abbildung 2.6b). Wie in einer Entwicklungsumgebung ermöglicht die Visualisierung einen 'go-to definition'-Befehl. Bei diesem wechselt die Ansicht in den Raum und zu der Stelle im Quellcode, an der das gewählte Element definiert wird.
Hori, Kawakami und Ichii (2019) fügen in ihrer VR-Visualisierung CodeHouse darüber hinaus eine Schnittstelle zu einem Debugger hinzu. So kann innerhalb der Visualisierung das Laufzeitverhalten des Software-Projekts nachvollzogen und auftretende Fehler innerhalb der Struktur lokalisiert werden.

Visualisierung von Software-Historie

Lanza (2001) schlägt vor, Software-Historie über sich verändernde Metriken zu visualisieren. Dazu entwickelte er eine "Evolution Matrix" (Abbildung 2.7a), in der eine Klasse zu einem Zeitpunkt als Rechteck dargestellt wird. Die Werte für Breite und Höhe werden durch zwei beliebige Metriken bestimmt. Um den zeitlichen Aspekt darzustellen, werden die Rechtecke in der Evolution Matrix angeordnet. Dabei enthält jede Zeile der Matrix eine Klasse des Software-Systems, die Spalten stellen die



(a) Evolution Matrix (Lanza, 2001)

(b) Spectrograph (Hassan, Jingwei Wu & Holt, 2005)

Abbildung 2.7. zweidimensionale, matrix-basierte Darstellung der Evolution der Software-Artefakte, basierend auf Metriken

verschiedenen betrachteten Zeitpunkte da. Betrachtet man die Matrix zeilenweise, ist die Evolution einer Klasse des Systems erkennbar. Die spaltenweise Betrachtung gibt einen Überblick über das System zu einem Zeitpunkt.
Einen ähnlichen Ansatz verfolgen Hassan, Jingwei Wu und Holt (2005) mit ihrem "Spectograph" (Abbildung 2.7b). Auch hier werden Artefakte des Software-Systems zeilenweise und Zeitpunkte spaltenweise angeordnet, die betrachtete Metrik wird durch die Farbe des Feldes kodiert. Hierzu werden vier Farben (grau, grün, gelb, rot) vorgeschlagen, die jeweils einem Viertel des Wertebereich der Metrik entsprechen.

Gall, Jazayeri und Riva (1999) visualisieren die Historie der Software-Architektur im dreidimensionalen Raum. Dazu wird die Software-Architektur zu einem Zeitpunkt in einem zweidimensionalen Graphen dargestellt. Durch Aufreihen dieser Graphen in der dritten Dimension wird die zeitliche Komponente eingebracht (Abbildung 2.8).

Im Gegensatz zu den bisher vorgestellten Arbeiten, bei denen Systemüberblick und zeitliche Veränderung in einer Ansicht kombiniert werden, trennen Beyer und Hassan (2006) diese Aspekte durch ein "Evolution Storyboard" (Abbildung 2.9). Sie stellen die Struktur der Software ähnlich wie Gall et al. (1999) als zweidimensionalen Graph dar. Anstatt die zeitliche Komponente durch die dritte Dimension zu visualisieren, werden die einzelnen Zeitpunkte hier auf getrennten Panels dargestellt. Durch diese

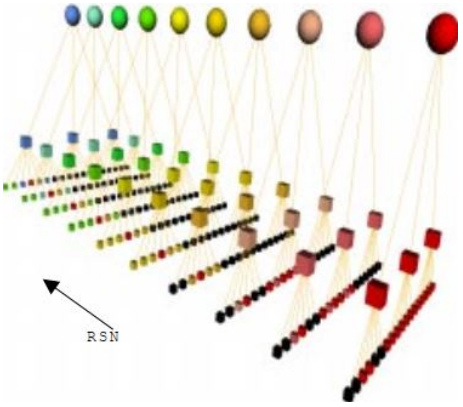


Abbildung 2.8. Darstellung der Software-Architektur als zweidimensionaler Graph. Die zeitliche Komponente wird durch Hinzunahme der dritten Dimension visualisiert. (Gall, Jazayeri & Riva, 1999)

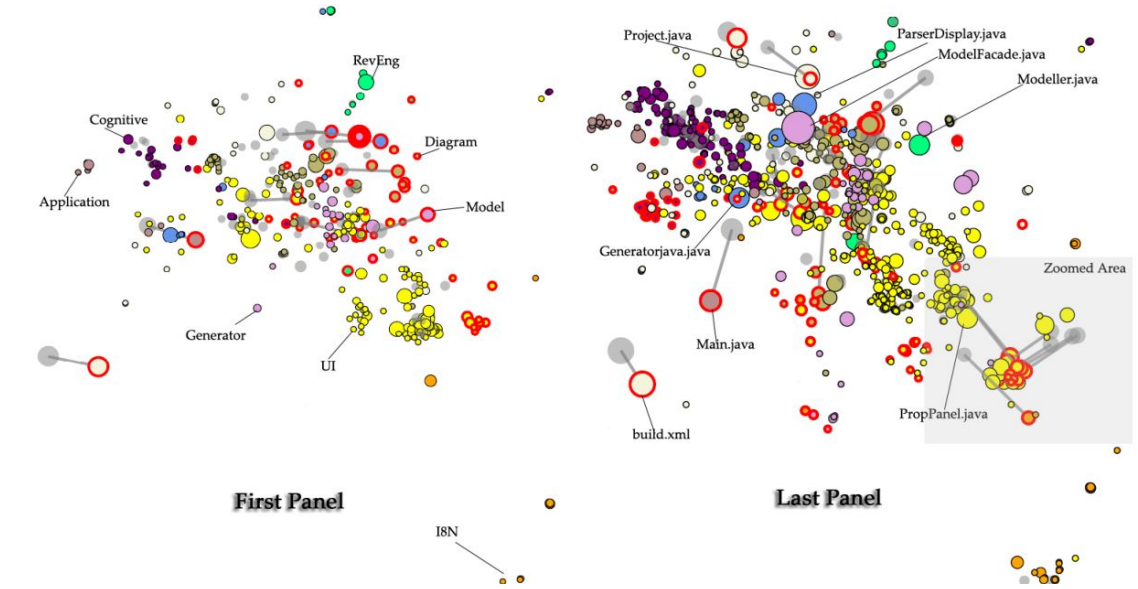


Abbildung 2.9. Software-Evolution als Storyboard (Beyer & Hassan, 2006)

Trennung haben Entwickler die Möglichkeit, einzelne statische Aspekte des Systems genauer zu untersuchen, während die Evolution durch Vergleich zweier nebeneinander liegender Panels sichtbar wird. (Beyer & Hassan, 2006)

Steinbrückner und Lewerentz (2010) entwickeln mit EvoStreets eine Visualisierung, bei der die Stadtmetapher auch die Software-Historie berücksichtigt. In dieser Arbeit wird die Paket-Hierarchie des Software-Systems durch Straßen realisiert, wobei untergeordnete Pakete als Seitenstraßen von der Straße orthogonal abzweigen, die ihr übergeordnetes Paket repräsentiert. Klassen entsprechen wiederum Gebäuden, die entlang der Straßen angeordnet sind. Die Größe der Gebäude kann dabei ebenfalls

den Wert einer Metrik repräsentieren. Um die Stadt entsprechend der Software-Historie zu verändern, stellen Steinbrückner und Lewerentz (2010) folgende Regeln auf:

- Neue Elemente (Klassen oder untergeordnete Pakete) werden am äußeren Ende der Straße angefügt, die das übergeordnete Paket repräsentiert. (z.B. Abbildung 2.10 R0 nach R100: P1 und P2)
- Die Straßenseite jedes Elements und die Reihenfolge der Elemente entlang einer Straße ändern sich nicht.
- Wird ein Element des Software-Systems entfernt, wird die zugehörige Visualisierung durch eine hellere Farbe als gelöscht markiert. (z.B. Abbildung 2.10 R0 nach R100: P4)
- Elemente, die innerhalb des Systems verschoben werden, werden behandelt, als wären sie an einer Stelle gelöscht und an anderer hinzugefügt worden.
- Muss aufgrund einer Änderung mehr Platz für ein Element geschaffen werden, werden alle an der Straße weiter außen liegenden Elemente verschoben. (z.B. Abbildung 2.10 R100 nach R200: P2)

Um die zeitliche Entwicklung stärker zu verdeutlichen, fügen Steinbrückner und Lewerentz (2010) EvoStreets eine Topologie hinzu: Der Zeitpunkt der Erstellung einer Klasse wird in ihrer Höhenlage codiert. Ältere Klassen befinden sich auf einer höhren Ebene als neuere Klassen. Hierdurch fallen die Straßen in EvoStreets tendenziell nach außen hin ab (Abbildung 2.11).
Zur Anreicherung der Karte mit Informationen schlagen Steinbrückner und Lewerentz (2013) vor, die Gebäudegrundrisse durch “property towers” zu ersetzten. Dabei kann ein Eigenschaftsturm aus mehreren gestapelten Segmenten bestehen. Diese Segmente können einerseits verschiedene Eigenschaften der Klasse anzeigen und deren Werte durch Form und Farbe codieren (Abbildung 2.12a (b)). Andererseits können die Segmente eine Eigenschaft der Klasse zu verschiedenen Zeitpunkten repräsentieren (Abbildung 2.12a (c) und (d)). Weitere Farb- und Formcodierungen der Gebäude sind möglich, um relevante Informationen aus dem Lebenszyklus des Software-Systems darzustellen, wie beispielsweise durch “authorship towers”, die zeigen, welcher Entwickler wann an den Klassen gearbeitet hat (Abbildung 2.12b)

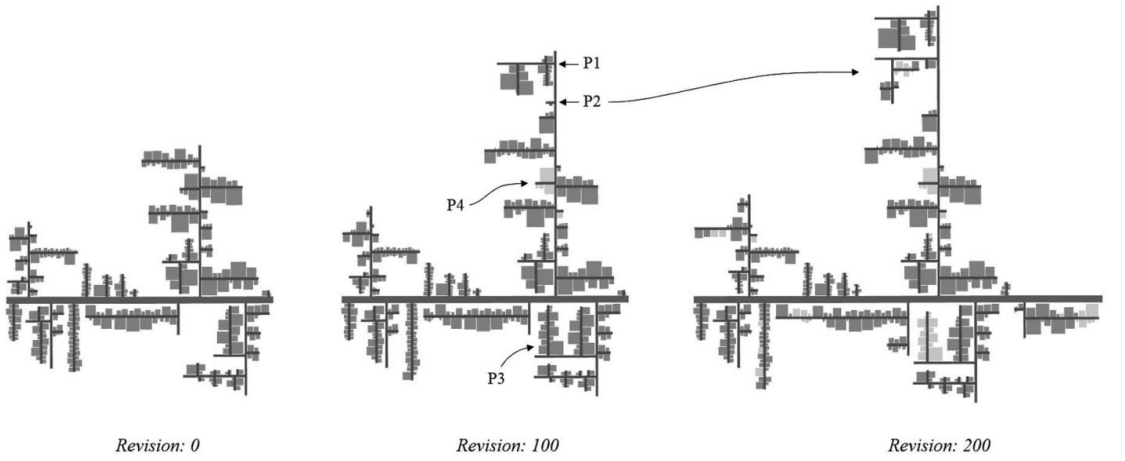


Abbildung 2.10. EvoStreets: Darstellung von Software-Evolution in einer Stadtmetapher (Steinbrückner & Lewerentz, 2013)

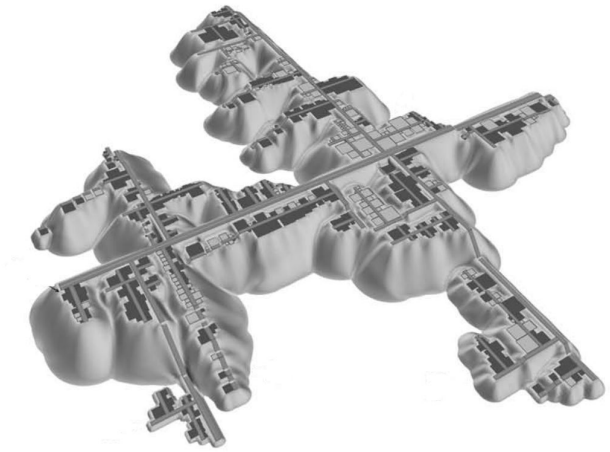
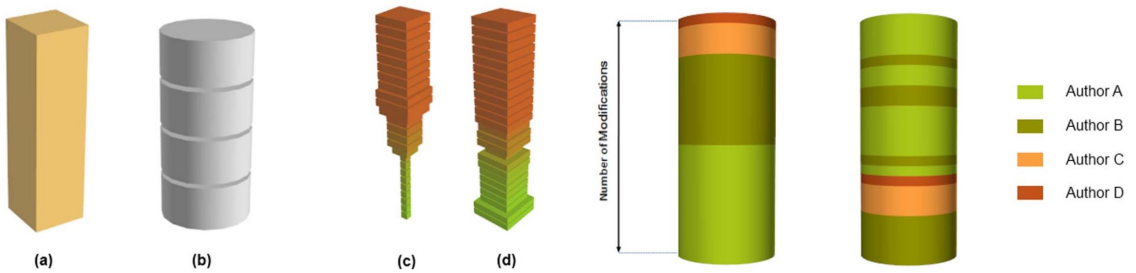


Abbildung 2.11. Stadtmetapher mit Topologie, die den Erstellungszeitpunkt der Klassen verdeutlicht. (Steinbrückner & Lewerentz, 2013)



(a) Eigenschaftstürme **(b)** Autorentürme

Abbildung 2.12. Erweiterte Gebäudeformen zur Darstellung von zusätzlichen Informationen (Steinbrückner & Lewerentz, 2013)

2.3 Einordnung

Die folgende Arbeit fokussiert sich auf die Visualisierung der Software-Historie von OSGi-basierten Software-Architekturen. Diese werden vor allem durch die beschriebene Gliederung in Bundles, Packages und CompilationUnits bestimmt. Die Inselmetapher von Misiak (2017) wird übernommen, nach der Bundles als Insel, Packages als Regionen und CompilationUnits als Gebäude dargestellt werden.
Hierdurch lässt sich das erstellte Konzept zur Darstellung von Software-Historie leicht in die Anwendung IslandVid (Schreiber, Nafeie, Baranowski, Seipel & Misiak, 2019) integrieren, da diese OSGi-Architekturen ebenfalls durch diese Zuordnung visualisiert.

Gemäß von Kurnatowski (2018) ist in der Historie von OSGi-Architekturen zu erwarten, dass Änderungen durch Hinzufügen, Entfernen und Verschieben von Elementen innerhalb der Gliederungsebene auftreten. Zur Vereinfachung wird ein Verschieben von Elementen hier nicht gesondert betrachtet, sondern als Löschen des Elements an einer Stelle und Neuerstellung an anderer Stelle aufgefasst.
Die Arbeiten von Beyer und Hassan (2006) und Steinbrückner und Lewerentz (2010) werden als Referenz bei der Erstellung des Konzepts herangezogen. Dabei wird vor allem der Erhalt der mentalen Karte übernommen, den beide Arbeiten berücksichtigen. Aus EvoStreets von Steinbrückner und Lewerentz (2013) wird außerdem die Idee einer Topologie und der Umgang mit gelöschten Elemente übernommen.

3 Anforderungsanalyse und Anwendungsfall

In diesem Kapitel werden die Anforderungen an das Konzept zur Visualisierung von Software-Historie durch eine Inselmetapher erarbeitet und zusammengestellt. Die Anforderungen ergeben sich aus verwandten Arbeiten und der Motivation für die Visualisierung, logische Abhängigkeiten identifizieren und klassifizieren zu können. Außerdem sollen Erfahrungen von Software-Entwicklern aus der Arbeit mit Software-Historie eingebracht werden.

3.1 Erhalt der Mentalen Karte

In den verwandten Arbeiten, die in Kapitel 2.2.2 vorgestellt wurden und ebenfalls Software-Historie repräsentieren, wird der Erhalt der mentalen Karte als zentrale Anforderung genannt ((Beyer & Hassan, 2006), (Steinbrückner & Lewerentz, 2013)). Der Begriff der mentalen Karte (engl.: mental map) wird von (Eades et al., 1991) im Zusammenhang mit dem Layout dynamischer Graphen entwickelt. Damit ist gemeint, dass bei der Darstellung eines zeitlich veränderlichen Graphen dessen grundlegende Geometrien möglichst unverändert bleiben soll: Knoten, die im Verlauf der Zeit erhalten bleiben, sollen auf ähnliche Weise relativ zueinander positioniert werden und bestenfalls ihre Platzierung kaum verändern. Dies ermöglicht dem Betrachter, die Übersicht, die er bereits zu Anfang über den Graphen gewonnen hat, weiter zu nutzen, ohne sich ständig neu orientieren zu müssen.

Im Bezug auf die Darstellung von Software-Historie durch eine Inselmetapher soll entsprechend gefordert werden, dass die Elemente, die Software-Artefakte darstellen, im zeitlichen Verlauf durch ihre Position zueinander identifiziert werden können. Die Ansicht zwischen den Zeitschritten sollte sich entsprechend nicht abrupt ändern. Diese Forderung gilt sowohl für die Übersicht über das System in Form der Inselwelt als auch für die Ansicht eines einzelnen Bundles und die damit Verbundene Struktur der Insel.

Anforderung 1: Die mentale Karte soll bei der Visualisierung der Software-Historie auf Systemebene erhalten bleiben.

Anforderung 2: Die mentale Karte innerhalb der Darstellung eines Bundles soll erhalten bleiben.

Die Visualisierung soll im Laufe des Entwicklungsprozesses mehrmals zur Unterstützung herangezogen werden soll. Zwischen diesen Nutzungen kann sich das betrachtete Software-System weiter verändern. Daher wird außerdem gefordert, dass die Positionierung der Elemente rekonstruiert werden kann, um den Effekt der mentalen Karte auch über längere Zeiträume aufrecht zu erhalten.

Anforderung 3: Die mentale Karte soll bei einem Neustart der Visualisierung weiterhin erhalten bleiben.

3.2 Identifizierung logischer Abhängigkeiten

In diesem Abschnitt wird die eingangs beschriebene Motivation zur Visualisierung von Software-Historie nochmals konkretisiert. Ziel ist es mithilfe der Darstellung logische Abhängigkeiten innerhalb des Projekts zu identifizieren. Diese Informationen können den Planungsprozess in der software-Entwicklung unterstützen. Aus diesem Anwendungsgebiet können ebenfalls Anforderungen abgeleitet werden.

3.2.1 Logische Abhängigkeiten

Logische Abhängigkeiten (Gall et al., 1998) bestehen “zwischen Quellcode-Dateien, die oft zusammen geändert werden, obwohl zwischen ihnen nicht notwendigerweise eine strukturelle Abhängigkeit besteht”. (Oliva et al., 2011).
Diese können einerseits als Hinweise für auftretende Fehler (Mockus & Weiss, 2000) dienen. Außerdem können sich in der Software-Historie Muster abzeichnen, dass Elemente oft gemeinsam geändert werden (Gall et al., 1998). Zimmermann, Weisgerber, Diehl und Zeller (2004) nutzen dieses Prinzip in einem Plug-in für die Entwicklungsumgebung Eclipse, das anhand logischer Abhängigkeiten Dateien vorschlägt, die wahrscheinlich im Zusammenhang mit einer gemachten Änderung ebenfalls angepasst werden müssen.

Gründe für logische Abhängigkeiten

Weiterführend identifizieren Oliva et al. (2011) in ihrer Arbeit verschiedene Gründe, wie es zu logischen Abhängigkeiten kommen kann, und leiten daraus verschiedene Konsequenzen ab:

1. **Refactoring elements that belong to a same semantic class**
   In diesem Kontext handelt es sich bei einer semantischen Klasse um eine Gruppe von Artefakten, die gemeinsam einer Funktionalität oder einer Rolle innerhalb der Architektur zugeordnet sind. Eine Überarbeitung der semantischen Klasse führt zu Änderungen in den zugehörigen Artefakten und damit zu einer logischen Abhängigkeit zwischen diesen.
   Ein einfaches Identifizieren semantischer Klassen verbessert die Anpassbarkeit der Software und unterstützt die Planung von Maintenance-Aufgaben, da die betroffenen Artefakte vorher bekannt sind.
2. **Structural dependencies on a changing semantic class**
   Dieser Fall ist eine Erweiterung des ersten Grundes. Hier hat die Überarbeitung der semantischen Klasse noch Nebeneffekte auf weitere, strukturell abhängige Artefakte, die ebenfalls geändert werden müssen, jedoch nicht zur eigentlichen semantischen Klasse gehören.
3. **Cross-cutting concerns**
   Querschnittliche Belange beziehen sich nicht auf den Hauptaspekt der Software, sondern z.B. auf Logging, Laufzeit-Kontrolle oder Parallelität, die über viele Module innerhalb des Systems verteilt sind. Ensprechend entstehen durch querschnittliche Belange logische Abhängigkeiten zwischen sehr vielen Artefakten.
   Logische Abhängigkeiten könnten helfen, Cross-Cutting concerns zu identifizieren, um diese in Aspekte zu kapseln und damit die Modularität des Systems zu erhöhen.
4. **Overloaded revisions**
   Es kommt auch vor, dass mehrere Änderungen zufällig oder beliebig gleichzeitig in die Versionsverwaltung commitet werden und hierdurch logische Abhängigkeiten bilden. Oliva et al. (2011) unterscheiden zwischen
   a) **Mulit-action revisions**, bei denen der Autor mehrere Elemente aus verschiedenen Gründen ändert bzw. um verschiedene Aufgaben zu erfüllen. (In der Commit-Message sind ggf. nicht alle Gründe genannt). Dieser Fall

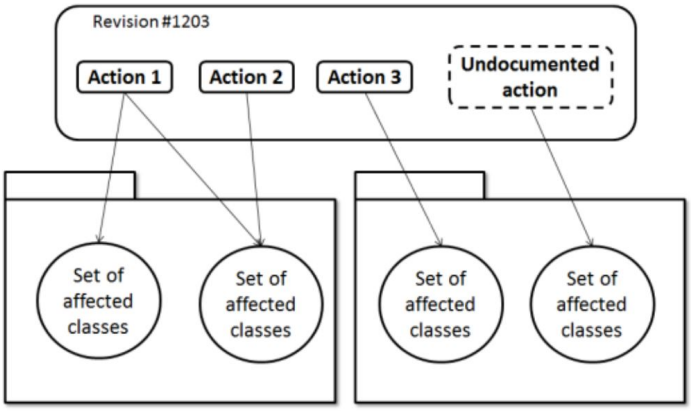


Abbildung 3.1. Beispiel für Multi-action revision (Oliva, Santana, Gerosa & Souza, 2011)

wird in Abbildung 3.1 illustriert.

b) **Convenience** In diesem Fall trifft der Autor beim Bearbeiten einer Aufgabe auf einen kleineren Fehler, der gemeinsam mit der eigentlichen Aufgabe korrigiert wird.

Overloaded revisions führen zu irreführenden Abhängigkeiten. Dadurch ergeben sich Ungenauigkeiten bei Hilfsmitteln, die notwendige Änderungen vorhersagen sollen (vgl. vorheriger Abschnitt, (Zimmermann et al., 2004)).

5. **Repository Operations**
   Hiermit sind Verschiebungen von Dateien in andere Ordner gemeint.

Daneben gibt es einige nicht kategorisierte logische Abhängigkeiten, wie z. B. das Formatieren von Code in mehreren Files.

Unter den oben dargelegten Gründen bietet besonders der erste Punkt, *"Refactoring elements that belong to a same semantic class"*, eine Möglichkeit zur effektiveren Arbeit am Software-Projekt: Durch das Wissen, welche Artefakte gemeinsam geändert werden müssen, um eine bestimmte Funktionalität anzupassen, ist es auch zukünftig einfacher, die Funktionalität weiter zu entwickeln. Jedoch müssen diese nützlichen logischen Abhängigkeiten effektiv von den Abhängigkeiten unterschieden werden, die durch die Punkte drei und vier entstehen.

3.2.2 Identifizierung in einer Inselmetapher

Zukünftig kann untersucht werden, ob sich eine Visualisierung von Software-Historie dafür eignet logische Abhängigkeiten zu erkennen und in einem Maße zu kategorisieren, das ausreicht, die alltägliche Planung von Aufgabenpaketen in der Software-Entwicklung zu unterstützen. Die Zuordnung der logischen Abhängigkeiten zu verschiedenen Kontexten soll dabei durch die menschliche Fähigkeit erfolgen, Muster in visuellen Darstellungen zu erkennen.
Da Funktionalitätsaspekte in der OSGi-Struktur bereits in einzelnen Bundles gekapselt sind und erweiterte Zusammenhänge durch Import-Beziehungen realisiert werden, wird angenommen, dass sich eine Inselmetapher gut für diesen Zweck eignen könnte.
Es wird erwartet, dass folgende, optische Auffälligkeiten identifizierbar sind:

1. Gemeinsame Änderungen von CompilationUnits,
   - die sich auf der selben Insel befinden,
   - die sich auf nahe beieinander liegenden oder stark verbundenen Inseln befinden,
   - bei denen es sich um ServiceComponents handelt, die über das Service-Layer verbunden sind,

   deuten auf die Überarbeitung einer semantischen Klasse hin.
2. Änderungen, die über das gesamte Archipel verteilte sind implizieren den Fall 3 *"Cross-cutting concerns"*.
3. Konzentrierungen der Änderungen auf mehrere, eher unabähngige Insel(-gruppen) könnte wiederum auf Fall 4a *Multi-action revisions* hinweisen. Hier könnten die unabhängigen Bereiche wieder als semantische Klassen begriffen werden.

3.2.3 Abgeleitete Anforderungen

Um die Änderungen einfach den jeweiligen Fällen zuordnen zu können sollten folgende Anforderungen in der Visualisierung erfüllt werden.

Anforderung 4: Hervorhebung geänderte Artefakte

Anforderung 5: Beziehungen zwischen den Bundles müssen einfach erkennbar sein.

3.3 Weitere Unterstützung der Software-Entwicklung

Neben Projektmanagern, die durch die Visualisierung der Veränderungen des Software-Projekts vor allem aus planerischer Sicht von dem zu entwickelnden Konzept profitieren, sind vor Allem auch Software-Entwickler am Projekt beteiligt. Im Rahmen eines Fokusgruppengesprächs soll herausgefunden werden, wie diese Gruppe mit Informationen aus der Software-Historie arbeitet und ob eine Visualisierung auch für sie hilfreich sein könnte. Für diesen Fall sollen funktionale Anforderungen erarbeitet werden.

Methodenbeschreibung Fokusgruppe

Bei einem Fokusgruppengespräch handelt es sich um eine moderierte Gruppendiskussion zu einer spezifischen Fragestellung. (Caplan, 1990). Das Gespräch profitiert durch gruppendynamische Effekte wie die Möglichkeit, auf die Meinungen der anderen Teilnehmer einzugehen, Argumente und Gegenargumente auszutauschen, sowie durch die verschiedenen persönlichen Erfahrungen, gemeinsam neue Ideen zu entwickeln.

3.3.1 Methode

Teilnehmer

Für eine Fokusgruppe werden Teilnehmer zusammengestellt, die einen ähnlichen Hintergrund haben, um eine gleichberechtigte Diskussion zu ermöglichen. Für die

Größe der Gruppe werden sechs bis zehn Personen empfohlen. Diese Anzahl ermöglicht eine ausreichende Dynamik zwischen den Teilnehmern einerseits und andererseits jedem Teilnehmer einen adequaten Anteil an Redezeit. (Caplan, 1990).

Entsprechend der Zielsetzung des Gesprächs sollen die Teilnehmer im Bereich der Software-Entwicklung beschäftigt sein. Daher wird die Fokusgruppe aus Mitarbeiter des DLR-Instituts für Simulations- und Softwaretechnik zusammengesetzt, die in der Software-Entwicklung arbeiten. Die Beschränkung der Teilnehmer auf diese Einrichtung ist aus organisatorischen und zeitlichen Gründen im Rahmen dieser Arbeit notwendig. Da die Anzahl der möglichen Teilnehmer sehr klein ist, kann allerdings keine gänzlich homogene Gruppe im Bezug auf Erfahrung und thematische Schwerpunkte zusammengestellt werden.
Es wurden sechs mögliche Teilnehmer persönlich für das Gespräch eingeladen.

Moderation

Die Moderation erfolgt durch die Verfasserin der vorliegenden Arbeit.

Ablauf

Der Leitfaden zum Fokusgruppengespräch befindet sich in Anhang A.

Gemäß dem Vorschlag von Caplan (1990) wird für den Beginn des Fokusgruppengesprächs zunächst eine Einführung geplant, die den Hintergrund des Gesprächs und organisatorische Fragen klärt. Anschließend wird zum Einstieg eine kurze Vorstellungsrunde (Block 0) vorgesehen, bei der die Teilnehmer gebeten werden, sich und ihre Erfahrungen im Bereich der Software-Entwicklung oder ähnlicher Bereiche kurz vorzustellen.

Bei der Strukturierung des anschließenden Gesprächs zur Erarbeitung von Anforderungen an eine Visualisierung von Software-Historie ist zu beachten, dass die Teilnehmer alle die Anwendung IslandViz bereits kennen. Die Art der Visualisierung in ihrem momentanen Stand wird von den Teilnehmern und deren Kollegen in Frage gestellt und vor allem als Forschungs- und Ausstellungsobjekt betrachtet.
Um den Gesprächsverlauf nicht zu früh durch diese Kontroverse zu blockieren, wird, wie ebenfalls von Caplan (1990) vorgeschlagen, ein Verlauf von allgemeinen Fragen zur Arbeit mit Software-Historie zu speziellen Fragen mit Bezug auf Visualisierung und eine Implementierung in IslandViz geplant.

Da das zu erstellende Konzept die Visualisierung von Software-Historie ermöglichen soll, wird in Block 1 betrachtet, inwiefern die Teilnehmer bereits mit Informationen aus der Historie arbeiten. Dazu wird erörtert, welche konkreten Fragestellungen im Alltag mit diesen Informationen gelöst werden. Hieran knüpft sich auch die Frage, welche Informationen für die Aufgaben benötigt und wie diese angewandt werden. Zusätzlich soll betrachtet werden, aus welchen Quellen die Teilnehmer momentan die benötigten Informationen erhalten. Aus den Antworten sollen Rückschlüsse gezogen werden, wie sich die Teilnehmer durch den zeitlichen Aspekt arbeiten. Es soll weiterhin ermittelt werden, welche Detailgrade im Bezug auf zeitliche Auflösung und Informationen zum Software-Projekt genutzt werden und daher in einer Visualisierung dargestellt werden sollten.

Um das Gespräch anschließend in Richtung Visualisierung zu leiten, wird in Block 2 diskutiert, an welcher Stelle bereits Visualisierungen im Alltag verwendet werden und für welche Aufgaben eine unterstützende Visualisierung gewünscht wäre. Hierbei soll auch konkretisiert werden, welche Aspekte jeweils in aktuell genutzten Visualisierungen dargestellt werden und zu welchen Aspekten zukünftig eine Visualisierung gewünscht wird.
Für den Fall, dass Teilnehmer keine konkreten Vorstellungen von verschiedenen Möglichkeiten für Visualisierungen haben, werden Bilder verschiedener bestehender Arbeiten (vgl. Kapitel 2.2.2) als zusätzliche Materialien bereit gehalten . Dieses vorgehen ist von Powel und Singel (1996) abgeleitet, die vorschlagen, zusätliche Impulse durch Materialien wie Bilder oder Filme zu setzen.

Der dritte Fragenblock fokussiert sich schließlich auf die geplante beispielhafte Implementierung der konzeptionierten Erweiterung in der Anwendung IslandViz. Dabei sollen vor allem Möglichkeiten zur Nutzerinteraktion mit der Software-Historie diskutiert werden. Dabei soll das in Kapitel 4.3 vorgestellte Konzept diskutiert werden und als Anregung für weitere Ideen dienen.
In diesem Block soll den Teilnehmern außerdem die Möglichkeit gegeben werden, eigene, konkrete Anforderungen an eine Visualisierung von Software-Historie zu stellen.

Das Fokusgruppengespräch endet mit der Möglichkeit für die Teilnehmer, weitere Anmerkungen zu machen.

3.3.2 Durchführung

Das Fokusgruppengespräch fand am 26. September 2019 von 13 bis 15 Uhr statt. Für das Gespräch wurde der Teamraum des DLR-Instituts für Simulations- und Softwaretechnik am Standort Köln-Porz verwendet. Die Plätze für die Teilnehmer waren, wie von Powel und Singel (1996) vorgeschlagen, in einem Dreiviertel-Kreis um einen Tisch herum angeordnet, um direkten Blickkontakt zwischen den Teilnehmern zu ermöglichen und so eine angenehme Gesprächsatmosphäre zu erzeugen. In der Mitte der Gruppe wurde ein Mikrofon der vorhandenen und in der Einrichtung bekannten Konferenztechnik platziert. Die Moderatorin saß innerhalb des Teilnehmerkreises auf einem der äußeren Plätze, um ebenfalls Blickkontakt mit allen Teilnehmern halten und gleichzeitig die Aufnahmetechnik unauffällig bedienen zu können.

Transkription

Die Transkription der Aufnahme des Fokusgruppengesprächs erfolgte nach dem einfachen Transkriptionssystem von Dresing und Pehl (2012). Dabei wurden unter anderem Wortverschleifungen an das Schriftdeutsch angenähert und Wortdoppelungen geglättet.
Die vollständige Transkription des Gesprächs befindet sich in Anhang B.

3.3.3 Ergebnisse

Soziodemographische Daten der Teilnehmer

Am Fokusgruppengespäch nahmen sechs Mitarbeiter des DLR-Instituts teil, die als Software-Entwickler arbeiten. Es handelte sich um zwei Frauen und vier Männer im Alter von 27 bis 42 Jahren. Die Erfahrung mit Arbeit an größeren Software-Projekten lag bei den Teilnehmern zwischen wenigen Monaten und ca. 15 Jahren. Alle Teilnehmer hatten während ihrer Studienzeit bereits Programmiererfahrung an kleineren Projekten oder akademischen Prototypen gesammelt.

Zusammenfassung des Fokusgruppengesprächs

Die Aussagen der Teilnehmer während des Fokusgruppengesprächs wurden in Themenbereichen zugeordnet. Die Bereiche, aus denen sich konkrete Anforderungen oder Schlussfolgerungen ableiten lassen, werden im folgenden beschrieben.

Verwendung von Software-Historie

Informationen aus der Software-Historie werden im Alltag der Software-Entwicklung für vier Aufgabengebiete herangezogen. Bei diesen handelt es sich um die Behebung von Fehlern, das Nachvollziehen von Änderungen für weitere Entwicklungsarbeiten, das Nachvollziehen von Änderungen aus planerischer Sicht und um Dokumentationsaufgaben.

Bei im Projekt auftretenden Fehlern kann die Software-Historie Hinweise auf deren Ursache geben. Zum einen kann es vorkommen, dass in bereits Funktionsfähigen Abschnitten unerwartete Fehler auftreten. In diesem Fall kann in der Historie zurück gegangen werden, um zu schauen, *"was hat sich denn da in letzter Zeit getan. Woran können so Seiteneffekte vielleicht entstehen"*, die den Fehler auslösen. Anderseits können Fehler bei einer neuen Funktional auftreten, die *"noch gar nicht lange drin ist und [...] deshalb einfach noch nicht fertig* ist.

Die Teilnehmer berichten, dass sie die Software-Historie verwenden, um sich über die aktuellen Entwicklungen und den Zustand des Projekts zu informieren. Dies reicht von einem allgemeinen Überblick *"was hat sich zuletzt getan"*, über das Nachvollziehen, wann und warum Aspekte hinzugefügt wurden, bis zur Vorbereitung auf eigene Aufgaben: *"Wenn ich in einen Code-Bereich reinschaue, wo ich lange nicht dran war [...], um [...] zu gucken, wann ist da zuletzt was passiert.Und [...] wer daran war."* Außerdem sei so der Kontext des Teilprojekts im Branch gegenüber des gesamten Projekts erkennbar (*Äuf welchem Merge-Stand der ist"*).

Neben diesen konkreten Anwendungsfällen berichtet ein Teilnehmer auch, dass er Informationen aus der Software-Historie auch für den Rückblick auf eine Aufgabe verwendent: *"Wenn ich weiß, jemand hat irgendwo gearbeitet, dann weiß ich, ein bestimmter Code-Bereich ist völlig klar, dass er daran gearbeitet hat, das ist ganz neuer Code, der da entsteht. Und dann will ich aber zum Beispiel schauen, hat er dafür andere Bereiche im System angefasst? Musste er irgendwo anders noch*

Schnittstellen erweitern oder anpassen." Eine Informationsquelle, die bisher mehr eine Idee ist, sei auch *"man könnt aus der Historie natürlich versuchen zu schauen, woran wurde parallel gearbeitet"*.

Quellen für Software-Historie

Die wichtigste Quelle zur Arbeit mit Software-Historie ist für die Teilnehmer das Versionsverwaltungssystem des Software-Projekts. Daneben werden Software-Projekte in der Abteilung auch über ein Mantis verwaltet. Darin können Issues angelegt und verwaltet werden. Einerseits beziehen die Teilnehmer aktuelle Information über die Software-Historie aus den zuletzt angelegten oder bearbeiteten Issues: *Üm sich auf den aktuellen Stand zu bringen, würde es natürlich auch helfen, einfach durch die letzten Issues zu gehen."*

Anforderung 6: Anzeige einer nach letzer Aktivität geordneten Liste von Issues.

Andererseits sind den Issues die Commits zugeordnet, in denen die betreffende Aufgabe bearbeitet wurde. Auf diesem Weg können relevante Zeitpunkte gefunden werden.

Anforderung 7: Möglichkeit, Commits nach zugehörigen Issues zu filtern.

Anforderung 8: Navigation zu einem Commit über den zugehörigen Issue.

Ausführlichere Erläuterungen zu Implementierugs-Entscheidungen im Entwicklungsprozess werden teilweise in Code-Kommentaren abgelegt. Außerdem werden auch Kollegen befragt, wenn es um das Nachvollziehen vergangener Entscheidungen geht: *"Manchmal ist es [...] 'weiches Wissen' einfach, was man dann [...] von Person zu Person weitergibt"*. Um den richtigen Ansprechpartner zu finden, ist die Kenntnis über den Autor des entsprechenden Commits notwendig.

Anforderung 9: Anzeige des Autors eines Commits.

Betrachtete Zeiträume

Die Zeitspanne innerhalb der Software-Historie, die von den Teilnehmern betrachtet wird, variiert je nach Anwendungsfall. Für die meisten der anfallenden Fragestellungen sind die vorherigen Arbeitstage, maximal die vergangenen 2 Wochen relevant: *Also meistens arbeiten wir [...] im Trunk oder [...] in Branches und die sind auch*

meistens [...] aktiv und dann geht es [...] seit dem letzten Arbeitstag [...] selten mehr als eins, zwei Wochen.

Anforderung 10: Schneller Zugriff auf Commits der aktuellsten zwei Wochen.

Wird ein Teilbereich betrachtet, der seid längerem nicht mehr verändert wurde, sind wiederum dessen letzte Commits von Bedeutung. Diese können jedoch auch mehrere Jahre zurück liegen.

Anforderung 11: Zugriffsmöglichkeiten auf alle Commits, unabhängig vom Zeitpunkt.

Anforderung 12: Schnelle Navigation vom angezeigten Commit aus zu Vorgängern und Nachfolgern.

User Interaktion mit Visualisierung von Software-Historie

Auf die Frage, welche expliziten Anforderungen die Teilnehmer an eine Visualisierung von Software-Historie stellen, wurden die Commit-Kommentare betont.

Anforderung 13: Anzeige der Commit-Kommentare.

Außerdem wird als praktisch empfunden, die Commits nach Branches Filtern zu können: *ich möchte jetzt gerne nur die Commits haben, die auf den Branch zutreffen".*

Anforderung 14: Möglichkeit, Commits nach Branches zu Filtern.

Genauso werden *"Filtermöglichkeiten [...] nach Autoren"* als hilfreich empfunden.

Anforderung 15: Möglichkeiten, Commits nach Autoren zu Filtern.

Im Zusammenhang mit der Bewegung entlang des Zeitstrahls, merkt ein Teilnehmer an: *"Die Steps bei so Commits, die können [einen] beliebigen Umfang haben"*, der von der Korrektur eines Buchstabens bis zum Einfügen neuer Strukturen und meherer 100 Zeilen Code reicht. In diesem Zusammenhang schlägt er eine proportionale Steuerung zur Navigation durch die Zeit vor: *"Das heißt, ich kann [...] Gas geben und schneller den Zeitstrahl lang gehen und wenn ich merke, ich komme in den richtigen Bereich, dann das Ganze langsamer wieder bewegen."*

Anforderung 16: Berücksichtigung der verschiedenen Umfänge von Änderungen bei der Navigation.

Anforderung 17: Schnelles Vorspuhlen ermöglichen.

3.3.4 Diskussion

Während des Fokusgruppengesprächs betonten die Teilnehmer öfter, dass es sich bei den betrachteten Informationen aus der Software-Historie um Änderungen auf Code-Ebene handelt, die in einer Entwicklungsumgebung am besten betrachtet werden können. Dabei meint ein Teilnehmer, dass er als Entwickler von einer möglichen Visualisierung *"ganz schnell wieder auf eine Code-Ebene"* zurückkommen wird.
Dieser Detailgrad wird in der geplanten Visualisierung nicht berücksichtigt. Daher werden Software-Entwickler nicht als primäre Zielgruppe der Visualisierung betrachtet.

Es fällt auf, dass ein Teilnehmer, der sich auch mit Aufgaben der Projektplanung beschäftigt, indirekt die bereits erarbeiteten logischen Abhängigkeiten erwähnte: *"Wenn ich weiß, jemand hat irgendwo gearbeitet, dann weiß ich, ein bestimmter Code-Bereich ist völlig klar, dass er daran gearbeitet hat, das ist ganz neuer Code, der da entsteht. Und dann will ich aber zum Beispiel schauen, hat er dafür andere Bereiche im System angefasst? Musste er irgendwo anders noch Schnittstellen erweitern oder anpassen."*

3.4 Zusammenfassung und Einordnung der Anforderungen

Bei den Anforderung 1 bis 3 handelt es sich um verschiedene Aspekte zum Erhalt der mentalen Karte. Dieser Themenbereich stellt den Schwerpunkt der Konzeption dar und wird auch in der Evaluation bewertet.
Auch die Hervorhebung von Änderungen entsprechend Anforderung 4 wird konzeptioniert und prototypisch umgesetzt werden.

Anforderung 5 bezieht sich auf die Visualisierung von Beziehungen im Software-Projekt. Dieses Bereich wird in der Arbeit nicht weiter thematisiert, da die statische Inselmetapher hierzu bereits Lösungen bietet.

Die Anforderungen aus dem Fokusgruppengespräch beziehen sich vor allem darauf, Metainformationen der Commits zugänglich zu machen. Autoren (Anforderung 9) und Commit-Kommentare (Anforderung 13) wurden dabei besonders hervorgehoben. Die Anforderungen 6, 7, 8, 14 und 15 beschreiben die gewünschte Funktionalität, Commits anhand ihrer Metadaten sortieren und filtern zu können und auf diesem Weg auf sie zuzugreifen. Um diesen Anforderungen gerecht zu werden, wird das Konzept Möglichkeiten zur Darstellung und Interaktion vorstellen, diese können aufgrund der zeitlichen Beschränkung jedoch nicht umgesetzt werden.

Schließlich beschreiben die Anforderungen 10, 11, 12 und 17 die Zugriffsmöglichkeiten auf Commits in Abhängigkeit zu ihrem Erstellungsdatum. In der vorgestellten Konzeption und Implementierung werden alle Commits als gleichwertig behandelt. Wodurch Anforderung 11 zunächst nicht besonders betrachtet werden muss. Eine lineare Navigation innerhalb der Software-Historie entsprechend Anforderung 12 wird implementiert, wohingegen die Möglichkeit, beschleunigt durch die Historie zu laufen (Anforderung 17), nicht vorgesehen wird.

4 Konzeption

Die Konzeption der Darstellung von Software-Historie durch eine Inselmetapher berücksichtigt zwei hauptsächliche Aspekte. Zum einen soll das Konzept sicherstellen, dass die mentale Karte des Nutzers erhalten bleibt während sich die Inselwelt entsprechend der Software-Historie verändert. Dies wird durch ein adaptives Layout der Regionen auf den Inseln und eine dynamische Positionierung der Inseln auf dem Meer erreicht. Zum anderen sollen die funktionalen Anforderungen zur Nutzerinteraktion mit den Aspekten der Software-Historie berücksichtigt werden.

4.1 Insel-Layout

4.1.1 Enhanced Hexagon Tiling Algorithm

In der Inselmetapher sollen Packages eines Bundles als verschiedenfarbige Regionen auf einer Insel dargestellt werden. Als Grundlage hierfür bietet sich der Enhanced Hexagon Tiling Algorithm (EHTA) von Yang und Biuk-Aghai (2015) an. Bei diesem handelt es sich um einen zufallsbasierten Algorithmus zur Darstellung von Informationen in landkarten-ähnlicher Form. Der EHTA erstellt dafür Regionen, indem er iterativ eine Menge von Zellen in einem Hexagon-Gitter auswählt.
Die Wahl der ersten Zelle der ersten Region geschieht zufällig. Von dieser aus werden der Region weitere Zellen zugeordnet. Diese werden wie folgt gewählt:

- Die nächste Zelle wird aus den noch nicht belegten Randzellen der Region gewählt, die an mindestens eine belegte Zelle angrenzen.

- Die Wahrscheinlichkeit, dass eine freie Zelle als nächstes gewählt wird, ist abhängig von der Anzahl der bereits belegten Nachbarzellen. Dabei ist $P(n)$, die Wahrscheinlichkeit, dass eine Zelle mit n belegten Nachbarn ausgewählt wird. Diese ist proportional zum Score s_n einer solchen Zelle.

$$P(n) \sim s_n = b^n \tag{4.1}$$

wobei der Faktor b bestimmt, wie kompakt die resultierende Auswahl erscheint.

Ist die notwendige Anzahl von Zellen einer Region festgelegt, wird die nächste Region am Rand der bereits bestehenden begonnen. Der Algorithmus legt so iterativ die benötigten Zellen aller Regionen fest. (Yang & Biuk-Aghai, 2015). Der EHTA enthält außerdem eine Backtracking-Komponente, falls begonnen wird, Zellen für eine neue Region in einem nicht ausreichend großen Loch zu belegen. Bei einem Loch handelt es sich in diesem Kontext um eine unbelegte Zelle/eine Menge von unbelegten Zellen, die vollständig von bereits belegten Zellen umgeben ist/sind.

4.1.2 Grundlage der Erweiterung

Auf den Regionen sollen entsprechend der Inselmetapher Gebäude platziert werden, die die CompilationUnits des Packages darstellen. Daraus folgt, dass die Fläche der Region proportional zur Anzahl der Gebäude sein muss, um diesen ausreichend Platz zu bieten. Diese Arbeit wählt die Zuordnung, dass auf jeder Zelle des dem Layout zugrunde liegenden Gitters genau ein Gebäude platziert werden kann.
Da sich die Anzahl der CompilationUnits innerhalb eines Packages über den Lebenszyklus der Software ändern kann, ändert sich auch die Anzahl der darzustellenden Gebäude innerhalb einer Region und dabei die benötigte Fläche.
Das Konzept zum Layout der Inseln muss daher vor allem sicherstellen, dass bei zunehmender Anzahl darzustellender Gebäude innerhalb der Region ausreichend Fläche zur Verfügung steht. Anforderung 3 beinhaltet, dass die Positionen der Darstellungen der Artefakte auch bei zeitlich auseinander liegenden Verwendungen der Visualisierung erhalten bleiben sollen. Zwischen diesen Nutzungszeitpunkten schreitet der Entwicklungszyklus voran und es können neue CompolationUnits entstehen.

Deshalb ist es nicht möglich, jeder Region bereits zu Anfang eine ausreichend große Fläche zuzuordnen, weil die notwendige Information noch nicht verfügbar ist, wie viele Zellen benötigt werden. Daraus ergibt sich, dass die Fläche der Regionen, und damit der Insel, im Laufe der Zeit zunehmen können muss.

4.1.3 Dynamisches Insel-Layout durch gesicherten Küstenzugang

Um zu ermöglichen, dass sich jede Region einer Insel bei Bedarf erweitern kann, wird das Insel-Layout so gestaltet, dass jede Region stets nicht belegte Randzellen hat, die zur Erweiterung belegt werden können. Im Weiteren werden freie Randzellen einer Insel im Sinne der Metapher als Küste bezeichnet (Abbildung 4.1).
Die präsentierten Ansätze wurden zur Illustration beispielhaft umgesetzt. Dabei wurden die Anzahl der pro Zeitschritt neu hinzukommenden Zellen innerhalb bestehender Regionen und die Anzahl neuer Regionen durch Zufallszahlen festgelegt.

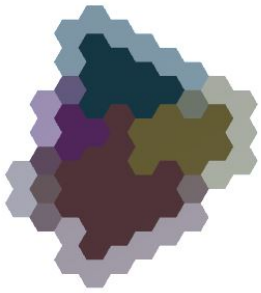


Abbildung 4.1. Insel aus belegten Zellen und Randzellen/Küstenzellen (heller)

Geometrischer Ansatz

Ein einfacher Ansatz, jeder Region einen Küstenzugang zu sichern, ist eine geometrischen Positionierung neuer Regionen auf den Achsen einer hexagonalen Grundform. Feste Richtungen und Breiten, in die sich die Regionen erweitern können, werden vorgegebenen. Wie in der Abbildung 4.2 zu sehen ist, bildet sich ein symmetrisches Layout. Es ist auch sichtbar, dass es bei großen Packages zu langen, schmalen Bereichen belegter Zellen kommt, die die Form der Insel wenig glaubwürdig erscheinen lassen. Außerdem ergeben sich gerade Grenzen zwischen den Regionen, was das

Insel-Layout insgesamt monoton und wenig überzeugend im Sinne der Landkarten-Metapher erscheinen lässt.

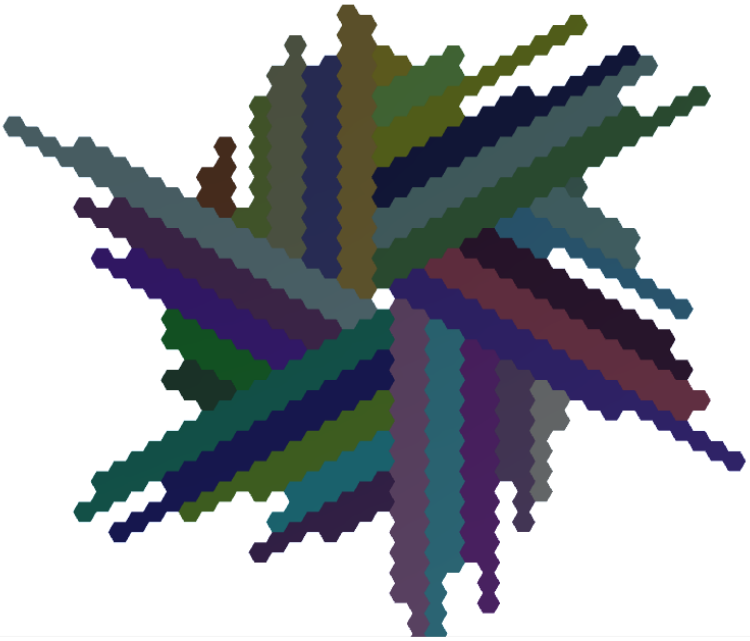


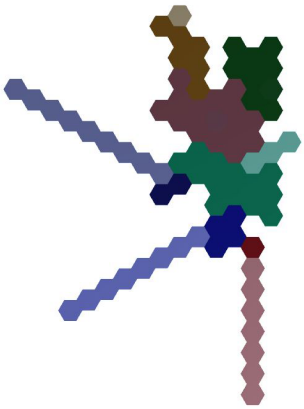
Abbildung 4.2. Symmetrisches Insel-Layout mit neuen Regionen auf den Achsen eines Sechsecks und zwei Zellen breiter Wachstumsmöglichkeit nach außen

Erweiterung des EHTA

Um einen interessanteren Grenzverlauf zu ermöglichen, wird auf den oben beschriebenen EHTA zurückgegriffen. Durch zusätzlichen Bedingungen bei der zufälligen Wahl der Zellen soll für jede Region einen Küstenzugang gesichtert werden.

Bevorzugte Küstenabschnitte

Zunächst wurde versucht, einzelne Zellen im Randbereich jeder Region als Wachstumsbereich frei zu halten. Tests am Prototyp ergaben jedoch, dass dieser Ansatz nicht ausreichend ist, um jeder Region einen dauerhaften Küstenzugang zu sichern. Die blockierten Zellen im Randbereich verhindern nicht, dass Nachbarregionen sich darum herum ausbreiten. So kann es vorkommen, dass ein ehemaliger Küstenabschnitt durch Wachstum der Nachbarregionen schließlich am Rand eines Lochs liegt und somit das unbeschränkte Wachstum der Region nicht mehr möglich ist.



(a) Insel mit Wachstumskorridoren der Regionen (helle Streifen)

(b) zum Vergleich: Wachstumskorridore ausgeblendet

Abbildung 4.3

Absolute Wachstumskorridore Um diese Probleme zu lösen, wird der Wachstumsbereich einer Region auf einen unendlich weit nach außen verlaufenden Wachstumskorridor erweitert. In Abbildung 4.3a sind die Wachstumskorridore einiger Regionen als hellerer Streifen dargestellt.
Bei diesem Ansatz wird, wenn eine neue Region erstellt werden soll, ihre erste Zelle so ausgewählt, dass von ihr aus der Wachstumskorridor ohne Überschneidung, mit bereits belegten Zellen oder Korridoren anderer Regionen, angelegt werden kann. Die Wahrscheinlichkeit $P_{start-opt}(n)$, dass eine Randzelle der Insel darauf überprüft wird, ob sie sich nach dem genannten Kriterium als Startzelle eignet, beträgt

$$P_{start-opt}(n) = b^6 - b^n \tag{4.2}$$

wobei n die Anzahl der bereits belegten Nachbarn der in Frage kommenden Zelle ist, bei b handelt es sich um die Konstante aus Gleichung 4.1. Regionen werden damit nach Möglichkeit an Zellen begonnen, die wenig stark von anderen Regionen eingeschlossen sind.
Ist eine überprüfte Randzelle nicht als Ausgangspunkt einer Region geeignet, werden weitere Zellen überpfrüft, bis eine Startzelle gefunden wurde. Dieses Vorgehen verhindert automatisch, dass Startzellen innerhalb von Löchern gewählt werden.
Zellen des Wachstumskorridors können nur durch die zugeordnete Region belegt werden, was auch verhindert, dass eine Region von ihren Nachbarn vollständig eingeschlossen wird.

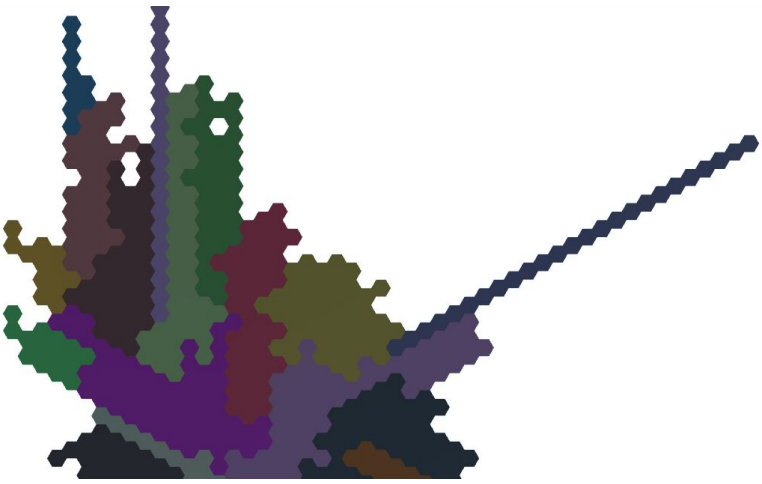


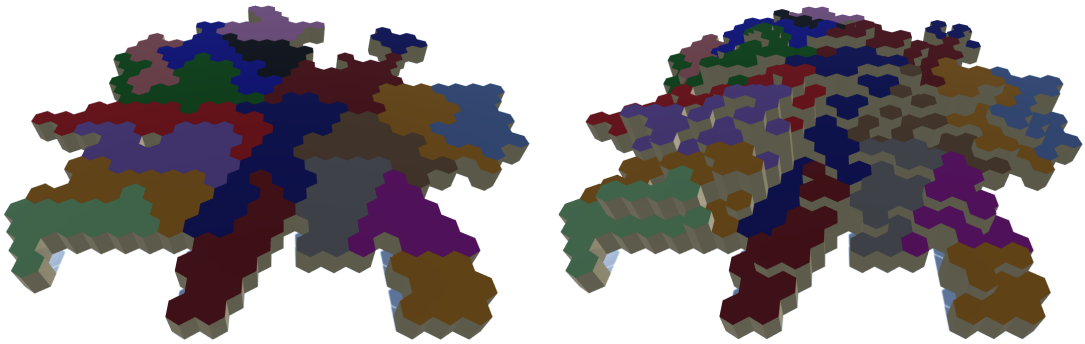
Abbildung 4.4. Beispiele für Regionen, die nur eine Zelle breit sind (violett, senkrecht nach oben & dunkelblau schräg nach rechts oben)

Verfeinerung

Wie in Abbildung 4.4 zu sehen ist, kann es wieder zu wenig realistischen Insel-Layouts kommen, wenn Regionen direkt zwischen zwei bestehenden Wachstumskorridoren angelegt werden und daher nur eine Zelle breit erscheinen können. Daher wird bei der Wahl einer möglichen Startzelle für eine neue Region der Abstand zu den nächsten bestehenden Wachstumskorridoren mit einbezogen. Die Formel 4.2 wird dazu folgendermaßen erweitert:

$$P_{start-opt}(n, step_r, step_l) = \begin{cases} 0.1 & \text{wenn } step_r = 0 \vee step_l = 0 \\ step_r * step_l * (b^6 - b^n) & \text{sonst} \end{cases} \tag{4.3}$$

Hierbei geben $step_r$ bzw. $step_l$ an, wie viele Zellen zwischen der betrachteten Zelle und dem nächsten Wachstumskorridor nach rechts bzw. links liegen. So erhöht sich die Wahrscheinlichkeit einer Zelle, als Startzelle in Betracht gezogen zu werden, wenn ihr Abstand zu bestehenden Wachstumskorridoren groß ist. Liegt die Zelle direkt neben einem Korridor ($step_r = 0 \vee step_l = 0$), beträgt die Wahrscheinlichkeit einen kleinen, konstanten Wert. So ist sichergestellt, dass auch eine Startzelle gewählt werden kann, wenn nur noch freie Randzellen neben Wachstumskorridoren zur Verfügung stehen.



(a) gleiche Höhe für alle Zellen **(b)** zeitabhängige Höhe jeder Zelle

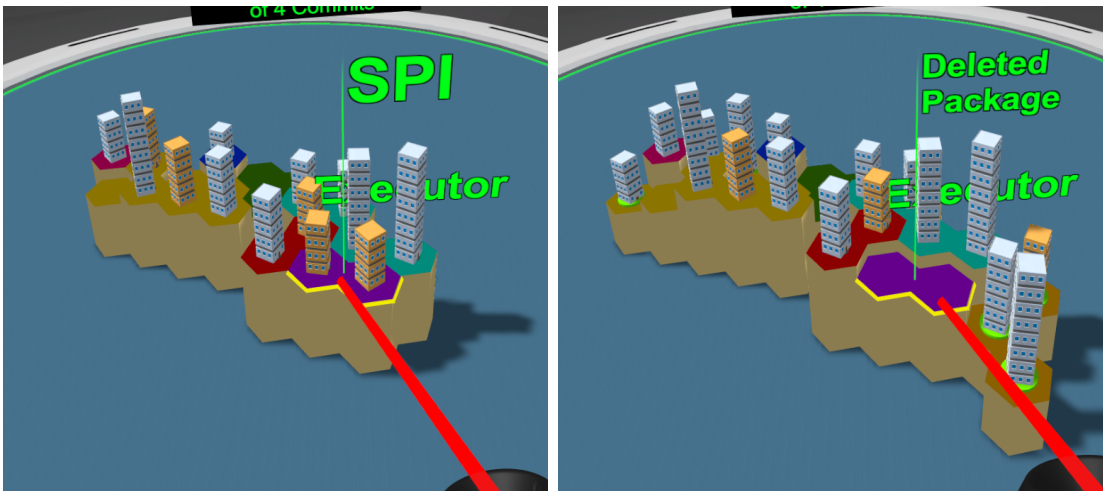
Abbildung 4.5. Küstenform und mögliches Höhenprofil der Insel

4.1.4 Höhenprofil und Küstenline

Nach dem Vorbild von Steinbrückner und Lewerentz (2010) soll das Insel-Layout zusätzlich durch eine Topologie angereichert werden: Die Höhe der einzelnen Zelle repräsentiert das Alter der zugehörigen CompilationUnit (Abbildung 4.5b). Dazu wird die Zelle einer neuen Klasse oder eines neuen Interfaces zunächst auf einer geringen Höhe über dem Meer dargestellt. Mit fortschreitender Zeit nimmt die Höhe der Zelle zu.
Daraus ergibt sich, dass sich Inseln im Laufe der Zeit aus der Meeresoberfläche erheben. Da innerhalb eines Packages ältere CompilationUnits häufiger im Zentrum der Insel und neuere am Rand positioniert werden, ergibt sich eine Kegelform. Im Sinne der Metapher könnte man von einer Vulkaninsel sprechen.

4.1.5 Löschen von Packages oder CompilationUnits

Wenn CompilationUnits innerhalb eines Packages gelöscht werden, wird kein Gebäude mehr dargestellt. Die Hexagon-Zelle die für die CompilationUnit reserviert wurde bleibt weiterhin teil der Region. Wird ein Package gelöscht ist dies gleichbedeutend damit, dass alle CompilationUnits des Packages gelöscht werden. Entsprechend können keine Gebäude mehr angezeigt werden. Die Fläche der Region wird weiterhin dargestellt, allerdings können keine Informationen zum Package mehr angezeigt werden. Anstelle des Namens des Packages wird die Region als “Deleted Package” bezeichnet (Abbildung 4.6).
Der Ansatz, die einmal zugeordneten Hexagonzellen nach dem Löschen der CompilationUnits nicht für neue Gebäude zu verwenden, soll den Erhalt der mentalen Karte



(a) Commit bevor das Package gelöscht wird

(b) Commit bei dem das Package gelöscht wurde

Abbildung 4.6. Auswirkungen auf das Insellayout beim Löschen eines Packages

unterstützen. Besonders beim Wechsel zwischen zwei Commits, die nicht aufeinander folgen kann so nachvollzogen werden, an welcher Stelle inzwischen Elemente verschwunden sind. Dies wäre nicht möglich, wenn die Flächen neu vergeben werden würden.

Wird die Insel mit der zeitabhängigen Topologie visualisiert, stellt die Höhe der Zelle einer gelöschten CompilationUnit dessen Lebensspanne dar: sobald das Gebäude entfernt wird, wächst die betreffende Zelle nicht weiter in die Höhe.

4.2 Insel-Positionierung

4.2.1 Graph-Layout durch einen Kräfteansatz

Bei einem Insellayout kann durch die Positionierung der Inseln auf dem Meer die Abhängigkeiten der Bundles implizit dargestellt werden. Dazu werden Bundles mit starken Beziehungen nahe beieinander platziert (Misiak, 2017).
Um ein solches Layout zu erstellen, eignen sich Algorithmen zum Zeichnen von Graphen, da die Bundles mit ihren Abhängigkeiten als ein gerichteter und gewichteter Graph aufgefasst werden können. Dabei werden die Bundles bzw. die sie repräsentierenden Inseln als Knoten aufgefasst und die Package-Import-Beziehungen als Kanten. Die Anzahl importierter Packages ergibt das Kantengewicht.

Eine häufig gewählte Heuristik ist der Kraftansatz von Eades (1984) für ungewichtete Graphen. Dieser betrachtet die Knoten eines Graphen als Partikel. Auf Knoten, die durch eine Kante verbunden sind, wirkt eine anziehende Kraft, die mit der Kraft einer Feder zwischen den Knoten verglichen werden kann. Zusätzlich herrscht zwischen allen Knoten eine abstoßende Kraft. Misiak (2017) erweitert die Formeln zur Bestimmung der Kräfte um die Berücksichtigung der Kantengewichte:
Die anziehende Kraft $\vec{F_a}$ zwischen zwei durch eine Kante verbundene Knoten beträgt

$$\vec{F_a} = c_1 * \vec{d_n} * \log \frac{d}{c_2} \tag{4.4}$$

Dabei handelt es sich bei $\vec{d_n}$ um den normalisierten Abstandsvektor zwischen den beiden Knoten, d ist dessen Betrag. c_1 kann als Steifheit der Feder interpretiert werden und ist ein festgelegter Faktor. Bei c_2 handelt es sich um die Länge der unbelasteten Feder. In diesen Faktor wird die Kantengewichtung einbezogen:

$$c_2 = (r_A + r_B) + c_3 * \frac{w_{max}}{w_E} \tag{4.5}$$

Bei r_A und r_B handelt es sich um die Radien der durch die Knoten repräsentierten Inseln. w_{max} ist das maximal im Graph auftretende Kantengewicht. Dies entspricht der maximalen Anzahl von Paketen eines Bundles, die ein anderes im Projekt importiert. Das Kantengewicht w_E der betrachteten Kante gibt an, wie viele Packages das betrachtete Bundle von seinem Nachbarn importiert.

Diese Formel bewirkt, dass die Gleichgewichtslänge der Feder, die die Kante mit den meisten Imports repräsentiert, am kürzesten ist. So führen große Importmengen zu einer stärkeren Anziehung und damit näheren Platzierung der Knoten.
Damit sich Knoten ohne gemeinsame Kante nicht zu nahe kommen oder überlappen, herrscht zwischen ihnen eine abstoßende Kraft.

$$\vec{F_r} = -\vec{d_n} * \frac{c_4}{d^2} \tag{4.6}$$

c_4 wird dabei so gewählt, dass sich die Knoten im Graph möglichst gleichmäßig über die vorhandene Fläche verteilen.
Der Algorithmus berechnet in mehreren Iterationen jeweils die auf die Knoten wirkenden Kräfte und verschiebt sie entsprechend.

4.2.2 Grundlage der Erweiterung

Das zu entwicklende Konzept soll ermöglichen, dass auch Software-Historie durch die Inselmetapher dargestellt werden kann. Die Historie beinhaltet auch, dass Bundles gelöscht oder neue erstellt werden. Außerdem können sich die Abhängigkeiten im Laufe der Zeit ändern. Auf die Metapher übertragen bedeutet dies, dass Inseln verschwinden oder hinzukommen. Außerdem soll die implizite Darstellung der Bundle-Abhängigkeiten durch nahe Platzierung (Misiak, 2017) wenn möglich beibehalten werden. Bei sich ändernden Beziehungen bedeutet dies auch, dass sich die Position der Inseln ändern können muss.

Dynamische Graphen

Dieses Problem ist als das Darstellen dynamischer Graphen bekannt. Als dynamischer Graph wird eine Folge von Graphen $G = [g_1, ..., g_n]$ bezeichnet, die unterschiedliche Zustände darstellen. Dabei sind Knoten und Kanten, die in mehreren Graphen der Folge auftreten eindeutig identifizierbar (Diel, Görg & Kerren, 2001).
Beim Layout dynamischer Graphen ist zusätzlich zu den Anforderungen an das Layout von klassischen Graphen (wie z.B. nach Fruchterman und Reingold (1991) gleichmäßige Verteilung der Knoten und wenige Kantenüberschneidungen) der

Erhalt der *mentalen Karte (mental map)* (Eades et al., 1991) eine zentrale Anforderung. Somit erscheinen bereits bestehende Algorithmen aus der Graphentheorie zur Darstellung dynamischer Graphen geeignet, um die Position der Inseln innerhalb der visualisierten Software-Hsitorie zu bestimmen.

4.2.3 Algorithmen zur Darstellung dynamischer Graphen

In der Literatur sind verschiedene Algorithmen zur Darstellung dynamischer Graphen erläutert. Zwei Ansätze werden hier aufgegriffen.

Aggregierter Graph

Dieser Ansatz berechnet die Darstellung der Graphen als ein globales Layout, von dem die Darstellung der einzelnen Graphen abgeleitet wird. Collberg, Kobourov, Nagra, Pitts und Wampler (2003) erstellen hierzu einen aggregierten Graphen. Dieser besteht aus allen Knoten und Kanten, die in der Folge des dynamischen Graphen vorkommen. Das Kanten- bzw. Knotengewicht gibt dabei an, in wie vielen Graphen der Folge die entsprechende Kante bzw. der entsprechende Knoten vorkommt.
Zunächst wird jedem Knoten im aggregierten Graph mithilfe eines kräftebasierten Ansatzes eine Position zugewiesen. Die hierfür verwendeten Formeln sind gemäß Collberg et al. (2003)

$$\vec{F_a} = \frac{w_e * d^2}{l^2} * \vec{d} \tag{4.7}$$

$$l = \frac{\sqrt{w_u * w_v}}{w_e} \tag{4.8}$$

$$\vec{F_r} = \frac{\sqrt{w_u * w_v}}{d^2} * \vec{d} \tag{4.9}$$

wobei es sich w_e bzw. w_v um die Kanten bzw. Knotengewichte handelt. l ist die optimale Federlänge der Kante e zwischen den Knoten u und v.
Nachdem so für jeden Knoten im aggregierten Graphen eine Position bestimmt wurde, werden diese bei der Darstellung der einzelnen Graphen der Folge für die Knoten verwendet.
Zur Verbesserung des Layouts der einzelnen Graphen der Folge schlagen Diehl und Görg (2002) außerdem vor, das auf dem Super-Graphen basierende Layout unter Berücksichtigung von Toleranzen zu verfeinern. Dazu definieren sie die mentale Distanz $\Delta(l1, l2)$ zwischen zwei Layouts l_1, l_2 zweier Graphen g_1, g_2 als die Summe der

Distanzen (z.B. euklidische Distanz) zwischen den Positionen der Knoten in beiden Layouts.

$$\Delta(l_1, l_2) = \sum_{v \in V_1 \cap V_2} dist(l_1(v), l_2(v)) \tag{4.10}$$

Für die Verfeinerung wird, nachdem die Knoten mit den Positionen des Super-Graphen initialisiert wurden, nochmals ein kräftebasierter Layout-Algorithmus angewandt. Das Layout wird jedoch nur übernommen, wenn die mentale Distanz zwischen dem neuen Layout und anderen Layouts der Folge kleiner als ein Toleranzwert δ ist.

Historienkraft

Chapanond, Krishnamoorthy, Prabhu und J. (2010) schlagen eine einfache Erweiterung des oben beschriebenen kräftebasierten Ansatzes vor. Dabei wird neben den anziehenden und abstoßenden Kräften zwischen Knoten eines Graphen eine zusätzliche Kraft zwischen Knoten in benachbarten Graphen der Folge eingeführt, die im Folgenden als Historienkraft bezeichnet wird. Auf jeden Knoten wirkt so eine Kraft durch seine Vorgänger (und ggf. Nachfolger) aus den anderen Graphen der Folge. Damit wird der Knoten von seinen anderen Positionen angezogen, und so die mentale Karte erhalten.
Chapanond et al. (2010) geben keine Empfehlung, wie die Historienkraft genau berechnet wird. Während der Implementierung wurde folgende Formel zielführend verwendet:

$$\vec{F_h} = \sum_{i=i}^{w} c_5 * c_1 * \frac{1}{2^{i-1}} * \vec{d_n} * \frac{d^2}{c_4} \tag{4.11}$$

$$\vec{d} = v_{t-i} - v_t \tag{4.12}$$

Als Abstandsvektor $\vec{d}$ wird hier der Abstand zwischen der aktuellen Position des Knotens v und seinem i-ten Vorgänger v_{t-i} betrachtet. c_5 beschreibt das Verhältnis zwischen der klassischen, anziehenden Kraft und der Historienkraft. Die Fensterbreite w bezieht sich auf die Empfehlung nach Chapanond et al. (2010), nicht nur die direkten zeitlichen Nachbarn für die Berechnung der Historienkraft zu verwenden, um eine bessere Positionierung zu erhalten.
Für diese Implementierung wird darauf verzichtet, auch die Positionen der Nachfolger eines Knotens miteinzubeziehen. Dadurch ist es möglich, das Layout aller

Graphen der Folge zeitlich geordnet jeweils einmal zu berechnen, da nur die Positionen der Vorgänger benötigt werden. Anderenfalls müssten mehrere Iterationen über die gesamte Folge der Graphen durchgeführt werden, um ein Ergebnis zu erhalten. Hierauf soll bei dieser Arbeit verzichtet werden, um die Zeit zum Erstellen der Visualisierung möglichst kurz zu halten.

4.2.4 Löschen von Bundles

Sobald ein Bundle gelöscht wird, enthält der dem Layout zugrunde liegende dynamische Graph für die folgenden Zeitpunkte keinen entsprechenden Knoten mehr. Der Algorithmus kann daher der Insel keine Position mehr zuweisen. Entsprechend wird die Insel an allen Commits, die nach dem Zeitpunkt des Löschens liegen, in der Visualisierung vollständig ausgeblendet.

4.3 Nutzerinteraktion

4.3.1 Grundlage der Erweiterung

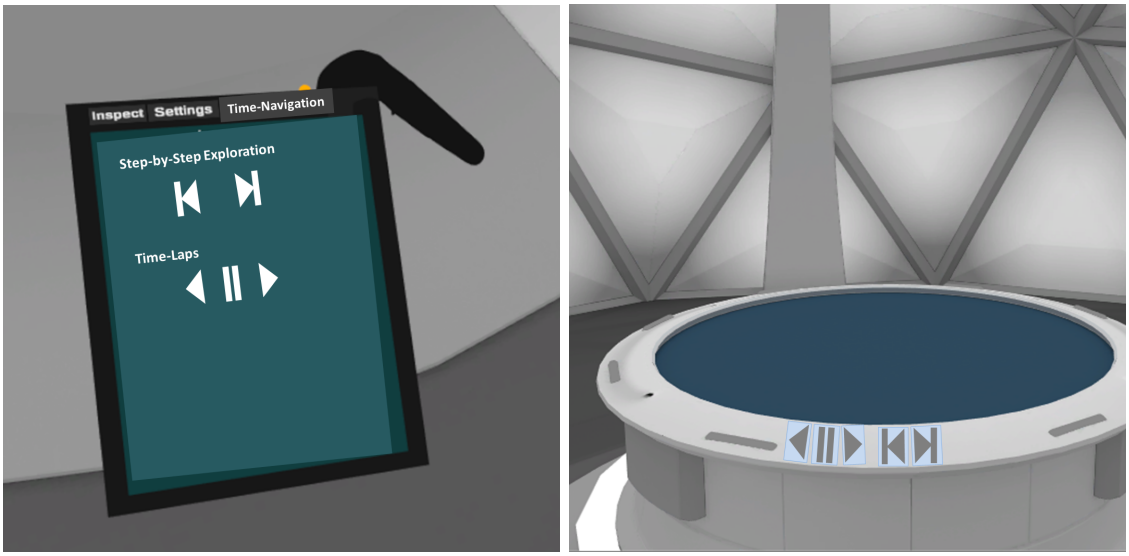
Da es sich bei der Anwendung, in die das Konzept beispielhaft integriert werden soll, um eine Visualisuerung in VR handelt, wird die Konzeption der Nutzerinteraktion spezifisch für diese Umgebung vorgenommen.
Die Interaktion des Nutzers mit der Visualisierung erfolgt über die zur VR-Brille gehörenden Controller. Der Nutzer kann die Ansicht der Inseln bewegen, vergrößern und verkleinern. Um Informationen zu den Inseln, Regionen und Gebäuden zu erhalten, können diese mithilfe eines virtuellen Laserpointers markiert werden. Die entsprechenden Informationen werden daraufhin auf einer Informationsfläche angezeigt. Steuerflächen in einem Menü, das am linken Controller eingeblendet werden kann ermöglichen das ein- und ausblenden von Abhängigkeitspfeilen und Beschriftungselementen sowie die automatische Navigation zu einer Gesamtansicht des Systems.
Die Möglichkeiten zur Nutzerinteraktion innerhalb der Software-Architektur zu einem Zeitpunkt sollen erhalten bleiben. Daraus ergibt sich, dass die Informationsfläche, das Menü und der Laserpointer als Auswahlwerkzeug erhalten bleiben.

Die folgende Konzeption stellt mögliche Schnittstellen vor, über die der Nutzer innerhalb der zeitlichen Dimension in der Software-Historie navigieren können soll. Weiterhin werden Möglichkeiten diskutiert, durch die zusätzliche Informationen angezeigt werden können.

4.3.2 Navigationsmöglichkeiten

Lineare Navigation entlang der Software-Historie

Das Konzept zur Nutzerinteraktion enthält als grundlegendes Element eine intuitive, lineare Navigation entlang der Historie. Dabei soll sowohl ein schrittweises Vorgehen als auch das Abspielen eines Zeitraffers möglich sein. Die Zeitlinie soll in beiden Fällen sowohl vorwärts als auch rückwärts erkundet werden können.
Es wird vorgeschlagen zur Eingabe entsprechender Befehle, Schaltflächen in der Anwendung zu platzieren. Dafür erscheinen die in Abbildnung 4.7 gezeigten Orte



(a) Menü **(b)** Rand des Tisches

Abbildung 4.7. Verschiedene Vorschläge zur Platzierung der Schaltflächen zur linearen Navigation innerhalb der Historie

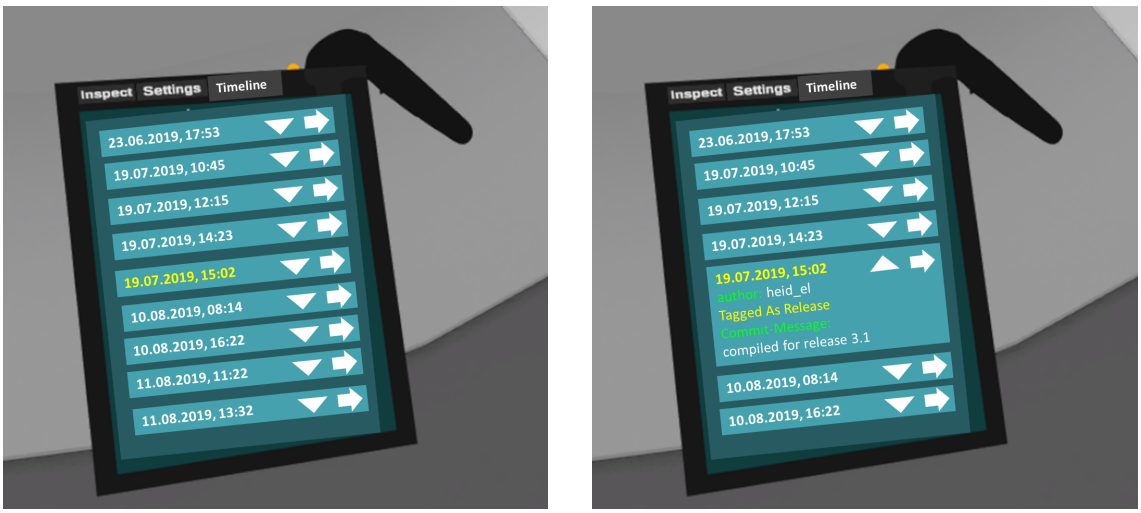
praktikabel: Eine Möglichkeit ist, das Menü um eine Seite “Time Navigation” zu erweitern, auf der die Schaltflächen angebracht werden (Abbildung 4.7a). Diese Variante ist konsistent zu den bereits vorhandenen Eingabemöglichkeiten. Eine andere Möglichkeit ist es, die Schaltflächen auf dem Rand des virtuellen Tisches zu platzieren. Dabei ist es notwendig die Position der Steuerelemente ständig der Position des Nutzers anzupassen, so dass dieser die Schaltflächen stets vor sich hat. Mit der zweiten Variante, wird eine neue Möglichkeit zur Nutzerinteraktion eingeführt. Diese verstärkt das Bild des interaktiven Tisches.
Die in den Abbildungen verwendeten Symbole sind an Steuerelemente bekannter Programme zum Abspielen von Mediendateien (z.B. Windows Media Player, VLC Media Player) angelehnt, um eine intuitive Navigation zu ermöglichen.

Gezielte Navigation innerhalb der Software-Historie

Das Fokusgruppengespräch (Kapitel 3.3.3) ergab unter anderem, dass Software-Entwickler bei der Betrachtung von Historie nicht nur direkt nebeneinanderliegende Commits vergleichen, sondern auch solche mit großer zeitlicher Distanz. Entsprechend soll die Anwendung auch ermöglichen, direkt zwischen Commits zu wechseln, ohne dafür z.B. mit dem Zeitraffer durch die Historie laufen zu müssen. Hierzu muss es möglich sein, direkt einen gewünschten Commit zu wählen. Für diese Funktion wird, wie in Abbildung 4.8a zu sehen, eine Erweiterung des Menüs vorgeschlagen.

Tabelle 4.1. Befehle zur gezielten Navigation zu Commits über das Menü

Beschreibung	Symbol
Einblenden zusätzlicher Informationen zum Commit	∨
Ausblenden zusätzlicher Informationen	∧
Anzeigen der Software-Architektur zum Zeitpunkt des Commits	⇒



(a) Liste der Commits **(b)** Liste mit zusätzlichen Informationen

Abbildung 4.8. Vorschlag zur gezielten Navigation innerhalb der Software-Historie durch eine Liste der Commits im Menü (Symbole siehe Tabelle 4.1)

Alle Commits werden nach Datum sortiert als Liste angezeigt. Im gezeigten Vorschlag wird der Commit über seinen jeweiligen Zeitstempelt identifiziert. An dieser Stelle ist es auch möglich, zusätzliche Informationen anzeigen zu lassen (Abbildung 4.8b) und bei der Visualisierung zum gewählten Commit zu wechseln (Funktionen und Symbole gemäß Tabelle 4.1).
Als Ergebnis des Fokusgruppengesprächs ist festzuhalten, dass im Gegensatz zu dieser konzeptionellen Darstellung bei einer Implementierung nicht das Datum herangezogen werden sollte. Das Kurzzeichen des Autors und die ersten Worte der Commit-Message eignen sich zur Identifizierung des gesuchten Commits besser.

4.3.3 Übersichtsdarstellungen

Neben den Möglichkeiten zur Navigation entlang der Zeitlinie erscheint es außerdem sinnvoll, eine zusätzliche Fläche zur Anzeige von Informationen zu schaffen. Diese könnte z.B. dafür verwendet werden, dem Nutzer eine Orientierung in der zeitlichen

Dimension zu bieten (Abbildung 4.9). Eine andere Möglichkeit wäre, die Informationen zum jeweils aktuell angezeigten Commit auf dieser Wand zu präsentieren (Abbildung 4.10). Die so dargestellten Informationen ergänzen die im vorherigen Abschnitt vorgeschlagene Liste von Commits im Menü, da es sich um die gleichen Informationen in anderer Darstellungsform handelt. Der Vorteil der Informationsfläche gegenüber des Menüs besteht darin, dass die Informationen dauerhaft präsent sind und dabei das Sichtfeld auf die Visualisierung der Architektur nicht eingeschränkt ist.

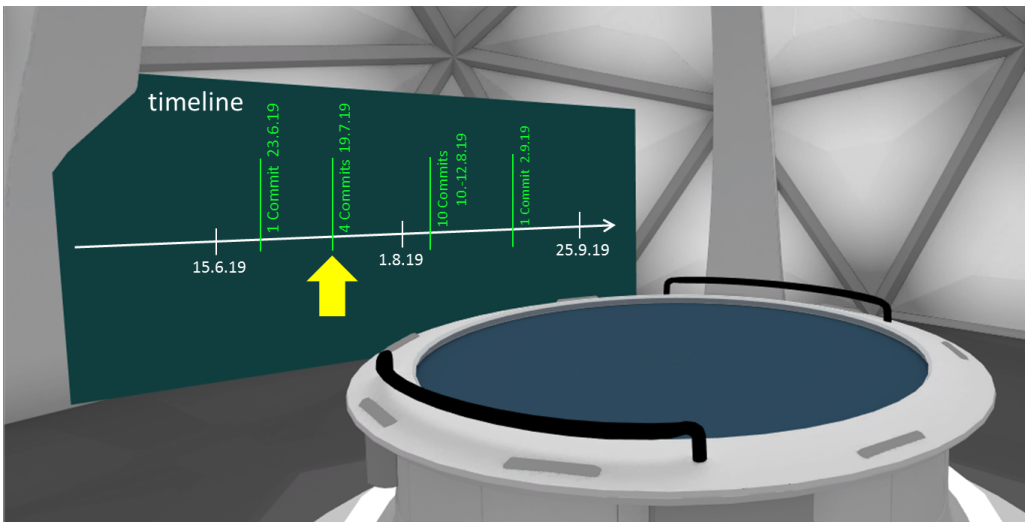


Abbildung 4.9. Informationsfläche zur Anzeige der Software-Historie. Zusätzliche Markierung des aktuell dargestellten Commits innerhalb des Zeitstrahls

Abbildung 4.10. Informationsfläche zur Anzeige der Metadaten des Commits

4.3.4 Hervorhebung von Änderungen

Um Änderungen einfacher erkennen und einordnen zu können, wird vorgeschlagen, die betreffenden Elemente farblich hervorzuheben. Diese Markierung soll sowohl bei einem Systemüberblick als auch in Detailansichten aussagekräftig sein.
Die bei der Hervorhebung berücksichtigten Änderungen am Software-System werden gemäß Tabelle 4.2 markiert. Das Konzept ist in den dort referenzierten Abbildungen zusätzlich illustriert.
Zusammengefasst spiegelt die Markierung der Insel wieder, ob diese neu ist, sich ihre Struktur verändert hat oder die Höhe eines Gebäudes verändert wurde. Ist eine Struktur- und eine Höhenänderung vorgenommen worden, wird die Insel entsprechend der Strukturänderung markiert.
Da Inseln, deren Bundles gelöscht wurden, nicht mehr in der Visualisierung vorkommen, kann in diesem Fall keine Hervorhebung stattfinden. Analog kann das Löschen eines Packages oder einer CompilationUnit nur auf System-Ebene hervorgehoben werden.

Änderung	Markierung der Insel	Markierung in der Detailansicht	Abbildung
Hinzufügen Bundle	grün	alle Gebäude grün	4.11
Hinzufügen Package	blau	alle Gebäude der neuen Region grün	4.12
Löschen Package	blau		
Hinzufügen CompilationUnit	blau	das neue Gebäude grün	4.12
Löschen CompilationUnit	blau		
signifikante Änderung der Größe einer CompilationUnit	weiß	das geänderte Gebäude blau	4.13

Tabelle 4.2. Konzept zur farbliche Hervorhebung von Änderungen an Inseln und Gebäuden

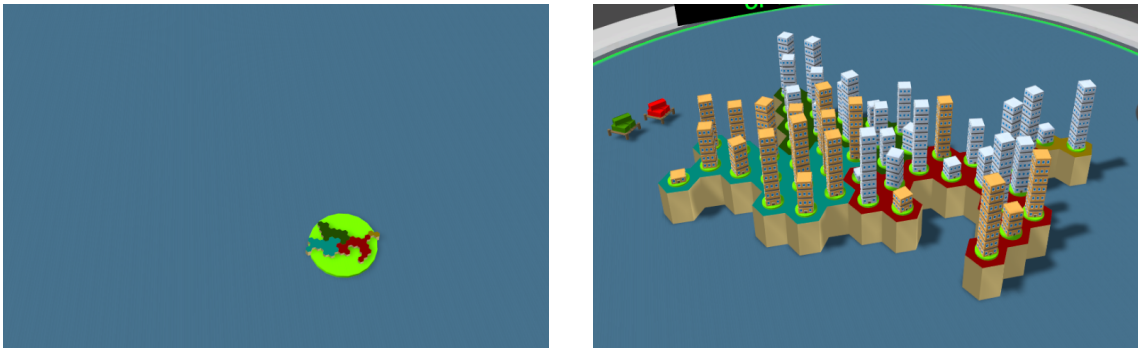


Abbildung 4.11. Hervorhebung einer neuen Insel

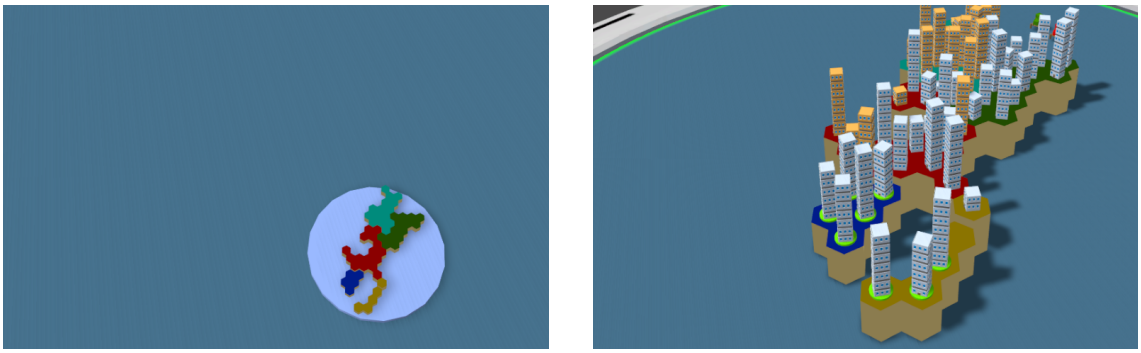


Abbildung 4.12. Hervorhebung struktureller Änderungen

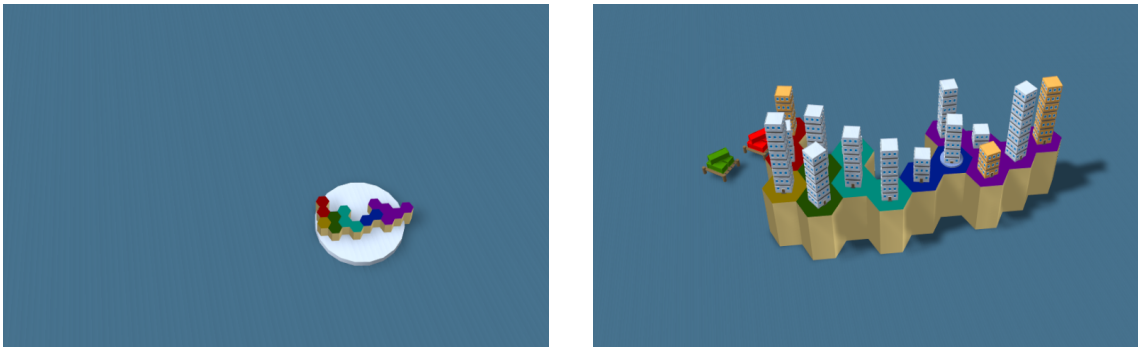


Abbildung 4.13. Hervorhebung der Insel, wenn sich nur die Höhe von Gebäuden ändert.

5 Implementierung

Die im vorherigen Kapitel erarbeiteten Konzepte zur Darstellung von Software-Historie durch eine Inselmetapher werden zur Demonstration in die Anwendung IslandViz integriert. In diesem Kapitel werden die Anwendung und der zum Testen gegebene Datensatz zunächst als Ausgangspunkt der Implementierung vorgestellt. Es folgen Erläuterungen zu ausgewählten Aspekten der Implementierung.

Als Entwicklungsumgebung für wurde Unity 2019.3 genutzt. Die Funktionalitäten wurden in C# implementiert. Eine HTC Vive Pro mit Controllern und Basisstationen wurde zur Verifikation der Umsetzung verwendet. Die Anbindung an das VR-Equipment erfolgte mit SteamVR 1.8.21. Die Entwicklung erfolgte auf einem Computer mit den folgenden technischen Daten:

- Prozessor: Intel(R) Xeon(R) CPU E-2650 v2 @2.60 GHz, 8 Kerne, 16 logische Prozessoren
- RAM: 64.0 GByte
- Grafikkarte: NVIDIA GeForce GTX 1080

5.1 Ausgangspunkt

5.1.1 IslandViz

Die Anwendung IslandViz wurde im DLR-Institut für Simulations- und Softwaretechnik zur Visualisierung von OSGi-basierter Software entwickelt. Diese glieder sich, wie in Kapitel 2.1.1 beschrieben, in Bundles, Packages und CompilationUnits. Beziehungen zwischen den Bundles werden durch Importe hergestellt, zusätzlich können Services definiert, implementiert und genutzt werden.

Misiak (2017) entwickelte IslandViz basierend auf einer Inselmetapher. Die gesamte Architektur wird als eine Insellandschaft auf dem Meer dargestellt. Die einzelnen Bundles werden dabei durch Inseln repräsentiert. Ihre Fläche ist in Regionen aufgeteilt, die den im Bundle enthaltenen Packages entsprechen. Die einzelnen CompilationUnits (Klassen und Interfaces) sind verschiedenfarbige Gebäude, die sich in den entsprechenden Regionen befinden. Die Höhe der Gebäude ist dabei proportional zu der Anzahl der Zeilen der Quellcode-Datei. Die Import-bedingten Abhängigkeiten zwischen Bundles werden durch Pfeile dargestellt (vgl. Abbildung 5.1). Dabei dienen die den Inseln vorgelagerten Häfen als Angriffspunkte der Pfeile.
Das Service-Layer wird durch ein weiteres Liniennetz (Abbildung 5.2b) dargestellt. Dabei können CompilationUnits in drei Rollen an einem Service beteiligt sein. Diese Rolle wird durch eine geometrische Figur gekennzeichnet, die über dem entsprechenden Gebäude schwebt. Bei den Rollen handelt es sich um ein Service Interface, das den Service definiert (blaue Kugel), Klassen, die den Service implementieren (grüne Kapsel) und Klassen, die den Service verwenden (roter Quader) (Abbildung 5.2a).

Die Visualisierung erfolgt in VR. Die virtuelle Umgebung besteht aus einem Raum mit unauffälligen Wänden und einem runden Tisch, auf dem die Inselwelt platziert ist. Die Navigation erfolgt durch Bewegung relativ zum virtuellen Tisch, sowie zoomen, drehen und schieben der Visualisierungsfläche auf dem Tisch. Außerdem können einzelne Elemente ausgewählt werden. Die dazu notwendigen Eingaben erfolgen über die zur Ausrüstung gehörenden Controller. In der Anwendung ist eine Informationsfläche integriert, die Informationen zu ausgewählten Elementen in Textform darstellt. (Abbildung 5.3). (Schreiber et al., 2019), (Misiak, 2017)

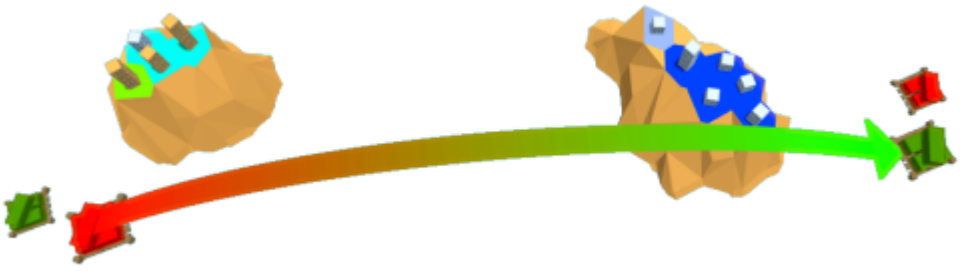
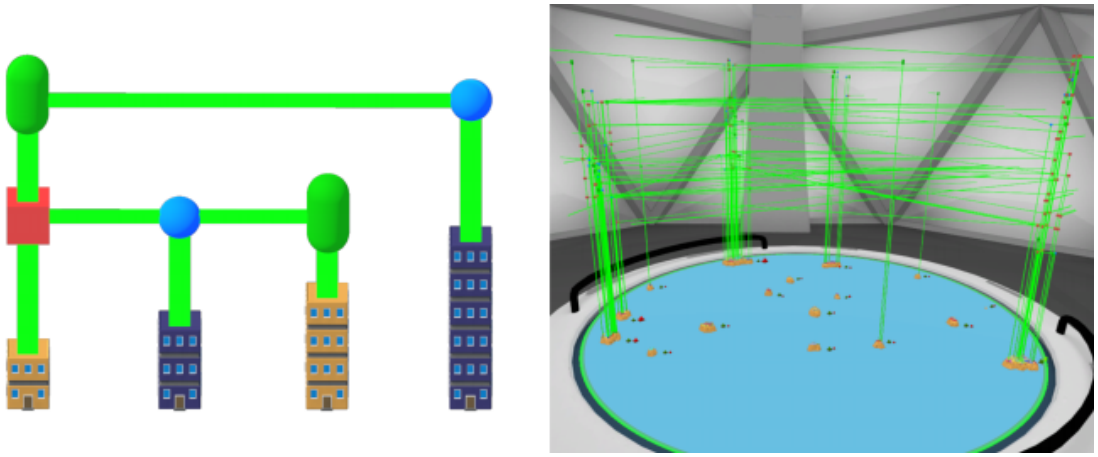


Abbildung 5.1. Visualisierung der Import-Abhängigkeiten zwischen Bundles (Schreiber, Nafeie, Baranowski, Seipel & Misiak, 2019)



(a) Elemente des Service-Layers mit Rollen **(b)** Visualisierung aller Services im Projekt

Abbildung 5.2. Visualisierung des Service-Layers (Schreiber, Nafeie, Baranowski, Seipel & Misiak, 2019)

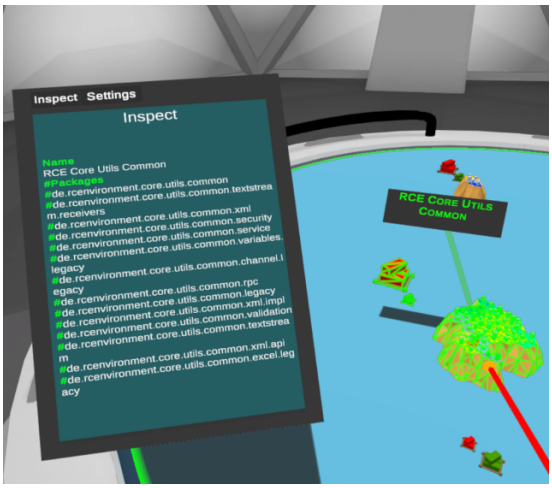


Abbildung 5.3. Informationen zur ausgewählten Insel in Textform (Schreiber, Nafeie, Baranowski, Seipel & Misiak, 2019)

5.1.2 Gegebenes Datenbankschema

Bei dem zum Testen der Implementierung gegebenen Datensatz handelt es sich um eine Neo4J Graphdatenbank. Das Vorgehen zur Extraktion der Software-Historie von OSGi-Projekten aus einer Versionsverwaltung und das zugehörige Datenbank-Schema wurden von von Kurnatowski (2018) entwickelt. Dieses Schema ist in Abbildung 5.4 zu sehen.
Die Entitäten *BundleImpl*, *PackageImpl*, *PackageFragmentImpl* und *CompilationUnitImpl* modellieren die durch OSGi bestimmte Strukturierung der Software-Artefakte. Die Entität *PackageImpl* dient der Repräsentation des Namensraums und bietet durch die Relation *PARENT_PACKAGE* eine weitere Verfeinerung der Hierarchie. *PackageFragmentImpl* stellt die konkrete Ausprägung im Bundle dar und wird daher auch in IslandViz als Region visualisiert.
Bei einer CompilationUnit kann es sich entweder um eine Klasse oder ein Interface handeln. Diese können aufeinander referenzieren (Relation *REFERENCES*) oder private, innere Klassen haben (Relation *NESTED_TYPES*).
Die Entitäten *ServiceComponentImpl* und *ServiceImpl* beziehen sich auf das OSGi-spezifische Service-Layer (siehe Kapitel 2.1.1). Bei einem Service handelt es sich um ein Java-Interface, das durch die Kombination der Entitäten *ServiceImpl* und *InterfaceImpl* sowie der Relation *IMPLEMENT* dargestellt wird. Services werden durch *ServiceComponentImpl* implementiert (*PROVIDED*) oder verwendet (*REFERENCED*).
Die Entitäten *ExternalTypeImpl*, *ConstructorImpl* und *FieldImpl* bieten die Möglichkeit, weitere Analyseergebnisse zur Struktur abzulegen.
Die Software-Historie wird durch die Entität *CommitImpl* abgebildet. Dabei repräsentiert jeder Knoten dieser Entität einen Commit im Versionskontrollsystem. Darunter hängt jeweils der gesamte Stand der Architektur entsprechend des jeweiligen Zeitpunkts. (von Kurnatowski, 2018)

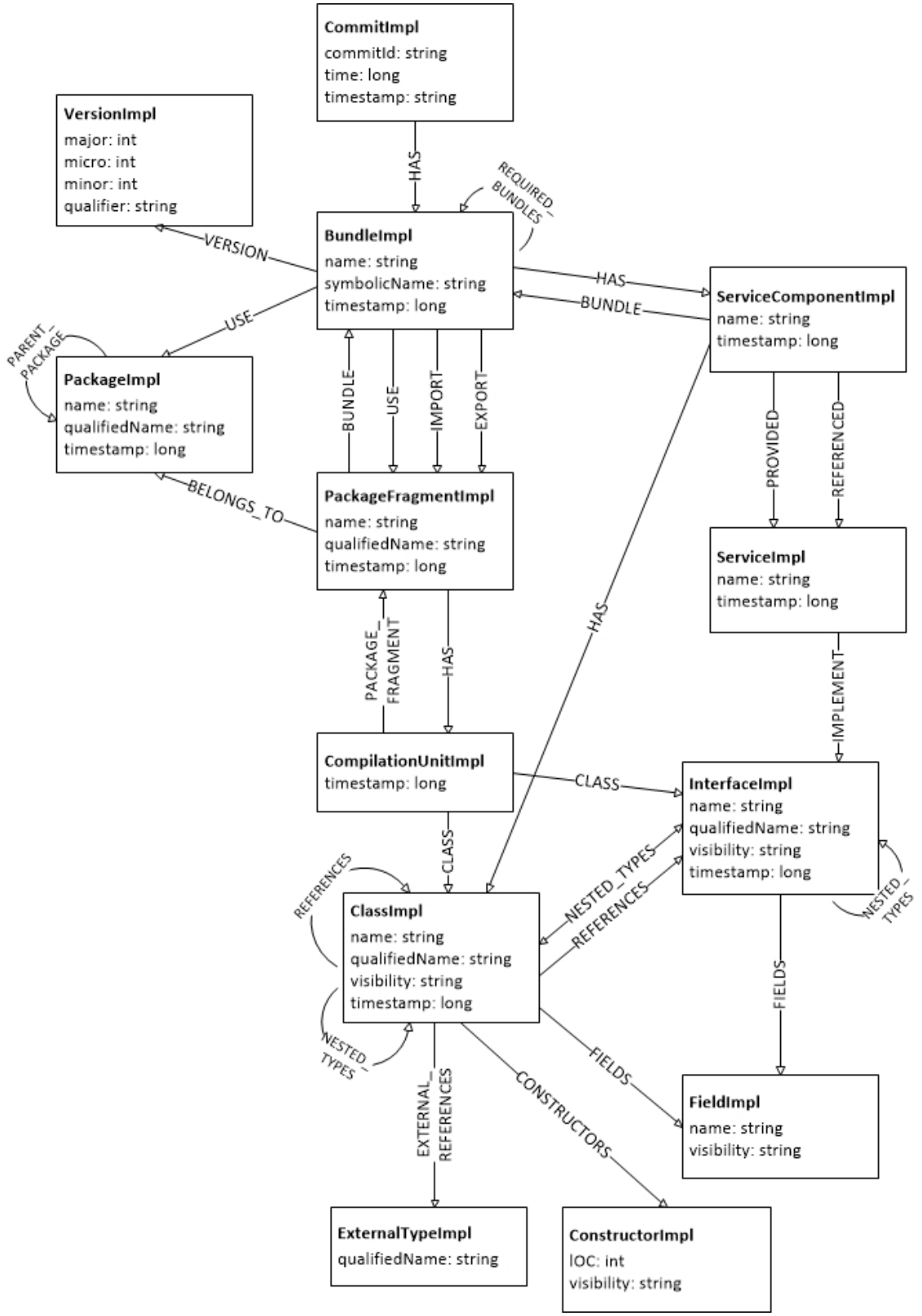


Abbildung 5.4. Datenbank-Schema der gegebenen Neo4J Graphdatenbank (angelehnt an (von Kurnatowski, 2018))

5.2 Nutzerschnittstelle zur Historie

Damit der Nutzer die Möglichkeit hat innerhalb der Visualisierung durch die Software-Historie zu navigieren und sich zu orientieren, wurde die virutelle Umgebung um entsprechende Schaltflächen und Informationstafeln erweitert.

Das Umschalten zwischen dargestellten Commits erfolgt über Pfeiltasten, die sich vor dem Nutzer auf dem Rand des Tisches befinden (Abbildung 5.5). Dabei besteht die Möglichkeit, die Software-Historie schrittweise vorwärts und rückwärts zu betrachten. Außerdem kann die Historie als Zeitraffer animiert dargestellt werden. Auch dieser kann in beide Richtungen der Zeitlinie abgespielt und beliebig angehalten werden. Die auf den Schaltflächen verwendeten Symbole sind in Tabelle 5.1 zusammengefasst.
Die Positionierung der Navigationselemente erfolgt dynamisch, sodass sich diese stets vor dem Nutzer befinden, wenn sich dieser um den Tisch herum bewegt.

Tabelle 5.1. Verwendete Symbole zur Navigation entlang der Software-Historie

Beschreibung	Symbol
Nächster Commit	> |
Vorheriger Commit	| <
Zeitraffer vorwärts	>
Zeitraffer rückwärts	<
Zeitraffer stop	||

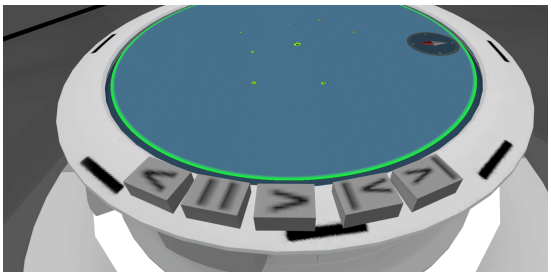
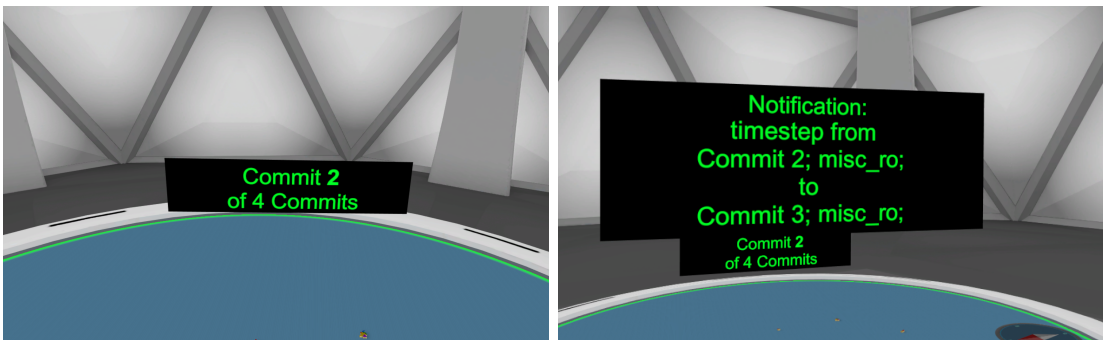


Abbildung 5.5. Schaltflächen zur Navigation innerhalb der Historie

Zur Orientierung wird eine Informationsfläche verwendet, die den Index des aktuell dargestellten Commits, sowie die Gesamtzahl der Commits im Projekt anzeigt (Abbildung 5.6a).



(a) aktuell angezeigten Commit **(b)** Übergang zum nächsten Commit

Abbildung 5.6. Informationstexte zur Orientierung in der Historie

Beim Übergang von einem Commit zum nächsten bzw. vorherigen erscheint über dem Tisch eine Informationstafel (Abbildung 5.6b). Diese zeigt sowohl für den zuvor dargestellten als auch für den neuen Commit dessen Index, das Kürzel des Entwicklers und den Beginn der Commit-Message an, sofern diese Informationen im Datensatz vorhanden sind.

5.3 Start der Anwendung

5.3.1 Überblick

Beim Start der Anwendung werden die in Abbildung 5.7 dargestellten Schritte nacheinander durchlaufen. Diese werden im Folgenden kurz zusammengefasst und bei Bedarf in den nächsten Abschnitten detaillierter erläutert.

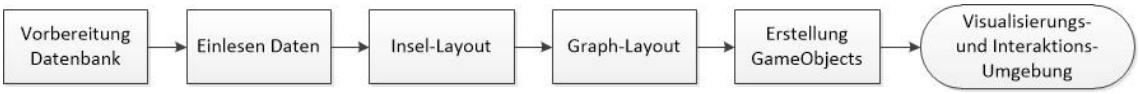


Abbildung 5.7. Ablauf beim Start der Anwendung

Vorbereitung der Datenbank

In diesem Schritt wird zunächst sicher gestellt, dass alle für die Visualisierung notwendingen Informationen über die Software-Historie in der Datenbank vorliegen. Nähere Informationen zu diesem Schritt werden in Abschnitt 5.3.2 dargestellt.

Einlesen der Daten

Die Informationen über Software-Architektur und -Historie des Projekts werden aus der Datenbank in eine interne Datenstruktur eingelesen, die in Abschnitt 5.4.1 erläutert wird.

Insel-Layout & Garph-Layout

Die Darstellung der OSGi-Komponenten wird entsprechend der in Kapitel 4.1 und 4.2 dargestellten Konzepte berechnet. Dafür werden zunächst die Layouts der Inseln durch den erweiterten EHTA ermittelt. Anschließend werden die Position der Inseln innerhalb der Darstellung durch den kraftbasierten Ansatz mit Historienkraft berechnet.
Um die mentale Karte auch zwischen den einzelnen Nutzungen der Anwendung aufrecht zu erhalten, werden die berechneten Informationen zur Darstellung in die Datenbank geschrieben. Dabei erhalten die Entitäten der Datenbank die folgenden zusätzlichen Attribute:

- *CommitImpl*: graphLayouted, islandsLayouted
- *BundleImpl*: posX, posZ
- *PackageFragmentImpl*: startCellX, startCellZ
- *CompilationUnitImpl*: cellX, cellZ

Wenn diese Attribute bereits vorhanden sind, erfolgt keine Neuberechnung. Die Layouts werden in diesem Fall aus den Daten rekonstruiert. Sind seit der letzten Nutzung neue Commits hinzugekommen, wird nur für diese die Darstellung neu berechnet.

Erstellung GameObjects

Im letzten Schritt des Ladevorgangs werden alle Unity GameObjects erstellt, die zur Darstellung der Inseln und ihrer Bestandteile notwendig sind. Abschnitt 5.4.2 beschreibt die Organisation der GameObjects genauer.

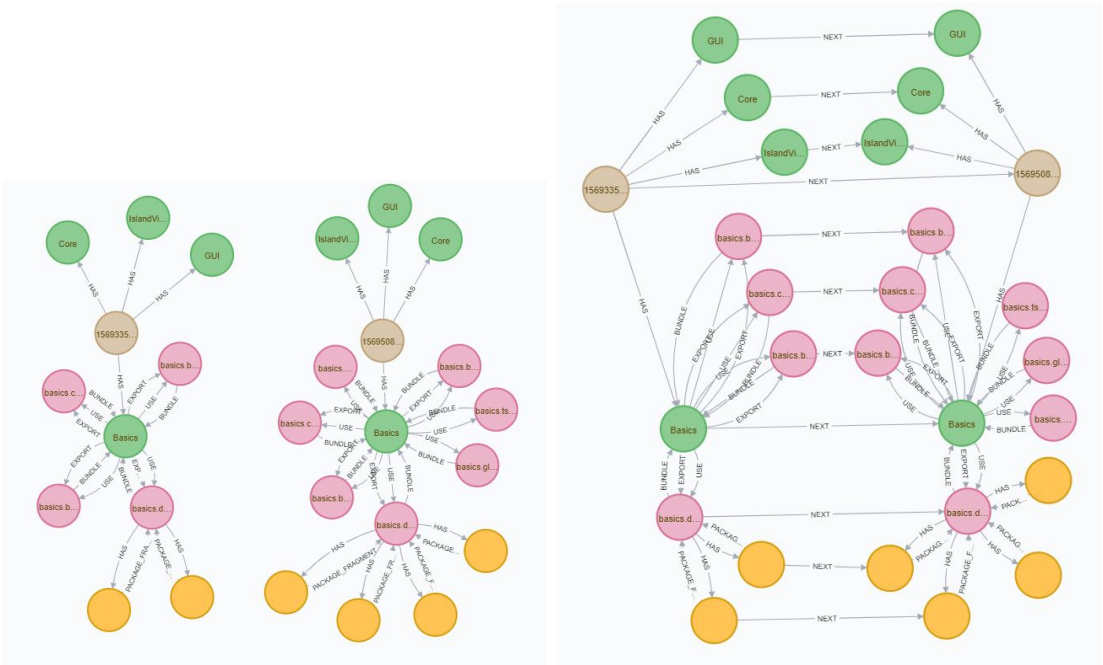
5.3.2 Vorbereitung der Datenbank

Die für die Anwendung zur Verfügung stehende Graphdatenbank besteht zunächst aus einzelnen, unabhängigen Graphen, die jeweils die Architektur an einem Zeitpunkt darstellen (vgl. Abbildung 5.4). Für die Implementierung ist es notwendig,

dass Entitäten miteinander assoziiert sind, die dasselbe Software-Artefakt zu verschiedenen Zeitpunkten darstellen. Hierzu werden zunächst die Commits jedes Branches in eine zeitliche Reihenfolge gebracht, wobei jeweils zwischen Vorgänger- und Nachfolgercommit eine *NEXT*-Beziehung hinzugefügt werden muss. Anschließend werden weitere *NEXT*-Beziehungen zwischen den entsprechenden Knoten mit den Labels *BundleImpl*, *PackageFragmentImpl* und *CompilationUnitImpl* eingefügt. So werden die Knoten, die dasselbe Artefakt zu aufeinander folgenden Zeitpunkten repräsentieren, verbunden und es entsteht für jedes Artefakt eine Zeitlinie, auf die in der Datenbank einfach zugegriffen werden kann.
Die Artefakte aufeinander folgender Commits werden stufenweise wie folgt assoziiert:

1. gleichnamige Bundles
2. noch nicht über den Namen zugeordnete Bundles, wenn möglich, über ähnliche enthaltene Packages
3. gleichnamige Packages zusammengehöriger Bundles
4. übriggebliebene Packages, wenn möglich, über ähnliche enthaltene CompilationUnits
5. CompilationUnits zusammengehöriger Packages durch ihren Namen

Die Erstellung der Relationen erfolgt durch entsprechende Anfragen der Anwendung an die Datenbank. Die notwendigen Namensvergleiche werden vollständig in der Datenbank ausgeführt. Die im Rahmen der Vorbereitung erzeugten Verbindungen bleiben auch nach dem Beenden der Anwendung in der Datenbank bestehen. Dieser Schritt wird demzufolge nur durchgeführt, wenn eine neue Datenbank verwendet wird oder neue Commits hinzukommen.
Abbildung 5.8 visualisiert die Veränderung beispielhaft, die während der Vorbereitung der Datenbank durchgeführt wird.



(a) Ursprungszustand **(b)** Hinzugefügte *NEXT*-Beziehungen

Abbildung 5.8. Ergebniss der Vorbereitung der Datenbank im Neo4j-Browser (braune Knoten: *CommitImpl*)

5.4 Interne Strukturen

5.4.1 Datenstruktur

Die innerhalb der Anwendung genutzte Datenstruktur wird in Abbildung 5.9 als Klassendiagramm dargestellt. Die Struktur bietet zum einen die Möglichkeit, Metadaten zum Projekt und zur Projekthistorie abzulegen. Das Zusammenspiel aus den Klassen *Commit*, *Issue*, *Author* und *Branch* bietet die Grundlage, um die Historie gezielt nach relevanten Zeitpunkten zu durchsuchen und zusätzliche Informationen zum Kontext des Commits zu erhalten. Diese Aspekte wurden während des Fokusgruppengespräch (vgl. Kapitel 3.3.3) als relevant identifiziert.

Die Klassen *Commit*, *Bundle*, *Package* und *CompilationUnit* entsprechen den Knoten *CommitImpl*, *BundleImpl*, *PackageFragmentImpl* und *CompilationUnitImpl* der Datenbank und speichern gemeinsam jeweils die Software-Architektur zu einem Zeitpunkt. Ebenfalls analog zur Datenbank enthält jede Instanz, die ein Software-Artefakt repräsentiert, Zeiger auf seine Vorgänger und Nachfolger. Es kann sich dabei jeweils um mehrere Instanzen handeln, wenn sich der zugehörige Commit an einer Verzweigung von Branches befindet.

Die Instanzen der Klassen *BundleMaster*, *PackageMaster* und *CompUnitMaster* fassen jeweils alle Objekte zusammen, die ein Softare-Artefakt zu verschiedenen Zeitpunkten darstellen. Diese Klassen sind außerdem die Schnittstelle zum Layout der Inseln, da während der Berechnung an dieser Stelle die Informationen über die Position im Hexagongitter abgelegt werden.

Eine Trennung der Klassen zwischen Artefakt und Artefakt-Master ist unter anderem notwendig, da sie den Komponenten der Unity GameObjects in verschiedenen Funktionen als Attribute dienen, wie im folgenden Abschnitt beschrieben wird.

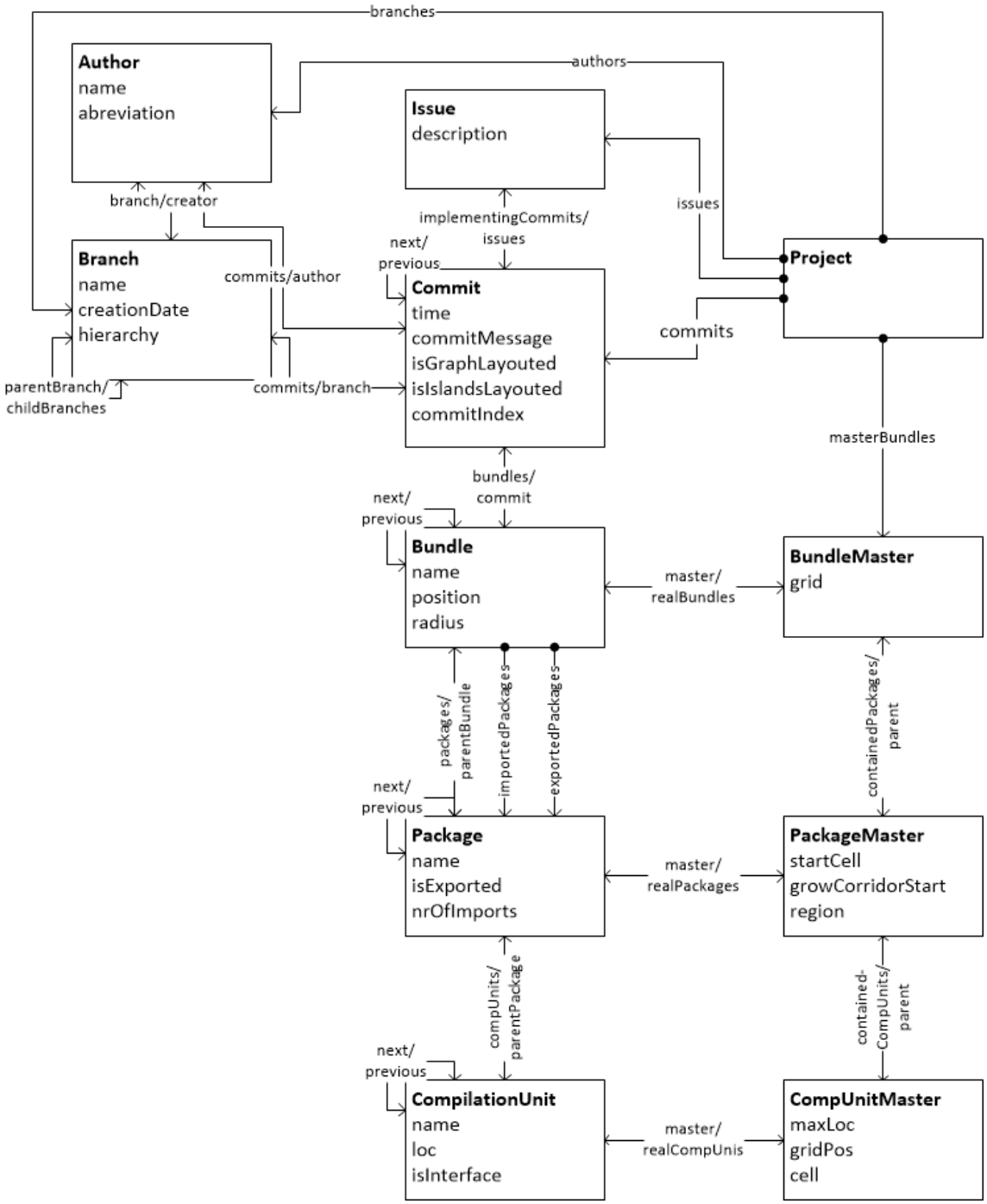


Abbildung 5.9. Klassendiagramm interne Datenstruktur

5.4.2 Unity GameObjects

Um eine Insel gemäß der Metapher in VR mit Unity visualiseren zu können, werden verschiedene graphische Elemente benötigt, die als einander untergeordnete Unity GameObjects realisiert werden. In der ursprünglichen Anwendung wurde jede Insel in ein Coastline-Gameobject, Dock-GameObjects, Region-GameObjects und Buildings unterteilt. Zur Erweiterung der Visualisierung für die Darstellung von Software-Historie sind zusätzliche Elemente erforderlich. Der resultierende Aufbau aus Unity GameObjects einer Insel ist in Tabelle 5.2 zusammengefasst, wobei auch die relevanten MonoBehavior-Komponenten der Objekte aufgeführt sind. Die Parent-/Child-Beziehung zwischen den GameObjects ist durch Einrückungen dargestellt.

Die gesamte Insel befindet sich in einem Container-Objekt, dessen Controller dafür sorgt, dass die Insel entsprechend der Historie des Bundles ein- und ausgeblendet wird und sie an die richtige Position verschiebt. Der Controller der Insel sorgt für die Anpassung aller untergeordneter Objekte entsprechend des anzuzeigenden Commits. Als Haupt-Attribut enthält er die Instanz der Klasse *BundleMaster* des Bundles, das die Insel darstellt. Die Komponente IslandGO ist aus der ursprünglichen Anwendung erhalten und dient der Interaktion mit der Insel. Das dazu notwendige Attribut ist die Instanz der Klasse *Bundle*, die die Informationen des zum aktuellen Commit gehörenden Bundles enthält.
Analog zur Insel ist auch die Region realisiert. Die Komponente Region dient der Interaktion und bezieht die dazu notwendigen Informationen aus der aktuellen Instanz der Klasse *Package*. Der RegionController aktualisiert die anzuzeigende Fläche im MeshFilter. Das dafür notwendige Attribut ist wiederum ein *PackageMaster*.
Die Höhe der Gebäude ist Abhängig von der aktuellen Anzahl der Zeilen einer CompilationUnit. Da es sich bei den Gebäuden mit unterschiedlicher Etagenanzahl um unabhängige Prefabs handelt, muss ein Building-GameObject ausgetauscht werden, wenn die zugehörige CompilationUnit ihren Umfang signifikant verändert. Daher ist wiederum ein BuildingContainer notwendig, der das Building-GameObjekt entsprechend der aktuellen Höhe ändert, oder löscht, wenn die zugehörige CompilationUnit im aktuellen Commit nicht mehr vorhanden ist. Der Nutzer kann durch die Komponente Building wiederum mit dem Gebäude interagieren.

Tabelle 5.2. Strukturierung der GameObjects innerhalb einer Insel

GameObject Bezeichnung	Relevante Script-Components
IslandContainer	IslandContainerController_Script
Island	IslandController_Script IslandGO
Coastline ImportDock ExportDock ChangeIndicator	
Region	RegionController_Script Region
BuildingContainer Building ChangeIndicator	BuildingController_Script Building

5.5 Ausblick

Die beispielhafte Implementierung realisiert die entwickelten Konzepte zum Erhalt der menalen Karte innerhalb der sich verändernden Inselwelt. Es sind außerdem grundlegende Möglichkeiten zur Interaktion mit der Software-Historie implementiert.

In weiteren Schritten kann zunächst der Umfang der möglichen Nutzerinteraktion erweitert werden. Besonders die Umsetzung der Anforderung 8 beliebigen Commits auswählen zu können ist eine noch ausstehende Funktionalität. Noch nicht implementiert sind außerdem die Anforderungen 7 und 15 Commits nach Metadaten filtern zu können. Ebenfalls fehlt eine Informationsfläche, auf der die Daten der Commits für den Nutzer angezeigt werden können.
Innerhalb der Datenstruktur sind bereits Klassen angelegt, in denen diese Informationen gespeichert werden. Damit besteht im Back-End bereits die Grundlage zur Umsetzung dieser Punkte.

Die momentane Anwendung visualisiert nur den Master-Branch. Zukünftig sollten auch die anderen Branches eines Software-Projekts visualisiert werden (vgl. Anforderung 14). Auch hier bietet die interne Datenstruktur bereits die Möglichkeit die entsprechenden Beziehungen zwischen den Commits abzulegen. Jedoch sind zur Darstellung von Branches noch einige Fragen zu klären. Diese sind unter anderem die

Umsetzung des Höhenprofils unter Berücksichtigung der Zweige und wie der Nutzer zwischen den Zeitlinien der Zweige navigiert.

Bei der technischen Umsetzung ist wahrscheinlich eine Optimierung sowohl der Layout-Algorithmen als auch der Datenstruktur notwendig, um große Software-Projekte mit vertretbarem Aufwand darstellen zu können.
In diesem Zuge besteht die Idee, die Berechnung der Darstellung von der eigentlichen Anwendung zur Visualisierung zu trennen und die Layout-Algorithmen in einem getrennten Programm auszuführen. Dieses könnte dann über Nacht, oder immer, wenn ein neuer Commit in der Versionsverwaltung gemacht wird, die Informationen für die Visualisierung vorbereiten. Die Visualisierung stünde so für den nächsten Arbeitstag oder ein Review zur Verfügung.

6 Evaluation

Die in diesem Kapitel beschriebene explorative Nutzerstudie soll zeigen, dass das im Rahmen dieser Arbeit erstellte und beispielhaft implementierte Konzept geeignet ist, Änderungen an einem Software-Projekt korrekt und effizient zu erkennen. Hierzu wird eine Between-Studie durchgeführt, die die ursprüngliche Anwendung IslandViz mit der Erweiterung vergleicht. Studenten der Universität Würzburg, die über Grundkenntnisse im Bereich Software-Entwicklung verfügen, lösen während der Studie Aufgaben in den Visualisierungen, wobei quantitative und qualitative Werte erhoben werden.
Es zeigt sich, dass der Erhalt der mentalen Karte in der Visualisierung zu einer korrekteren und schnelleren Bearbeitung führt. Das Hervorheben von Änderungen erhöht die Zufriedenheit der Nutzer, muss jedoch intuitiver gestaltet werden, um Fehlinterpretationen zu vermeiden.

6.1 Zielsetzung

In Kapitel 3.2.1 wurde dargestellt, dass das Wissen über logische Abhängigkeiten innerhalb eines Software-Projekts dabei helfen kann, zukünftige Aufgaben zu planen, und Bereiche zu identifizieren, die überarbeitet werden müssen. Hierzu müssen zunächst gemeinsam auftretende Änderungen innerhalb der Software-Historie erkannt werden. Anhand des Kontexts der Änderungen kann die logische Abhängigkeit anschließend klassifiziert werden. Besonders für die Klassifizierung der Abhängigkeiten erscheint die Visualisierung des Software-Projekts durch eine Inselmetapher geeignet.
Zunächst muss jedoch sicher gestellt werden, dass der Nutzer in einer solchen Visualisierung Änderungen identifizieren kann. Im nächsten Schritt kann untersucht werden, inwieweit die Visualisierung geeignet ist, um logische Abhängigkeiten zu klassifizieren.
Die in dieser Arbeit durchgeführte Nutzerstudie beschränkt sich auf den ersten

Aspekt: die Identifizierung von Änderungen mit Hilfe der Visualisierung.
Anhand der Anwendung IslandViz und der implementierten Erweiterung soll gezeigt werden, dass sich eine Visualisierung besser eignet, zeitliche Veränderungen aufzufinden, wenn sie die darauf optimiert wurde, die mentale Karte zu erhalten. Weiterhin soll untersucht werden, ob das Hervorheben der Änderungen in der Visualisierung einen zusätzlichen positiven Effekt auf die Arbeit des Nutzers hat.
Da es sich bei der Erweiterung zur Darstellung von Software-Historie um eine prototypische Implementierung handelt, sollen außerdem Möglichkeiten zur Verbesserung der Anwendung gesammelt werden.

6.2 Methode

6.2.1 Teilnehmer

Zur Anwerbung von Teilnehmern für die Nutzerstudie wird das Probandensystem der Universität Würzburg verwendet. Es wird nach Probanden mit Grundkenntnissen im Programmieren, Software-Entwicklung o.Ä. gesucht. Durch diese Bedingung sollen die Teilnehmer die für die Studie relevanten Eigenschaften von Projektverantwortlichen widerspiegeln, die als eigentliche Zielgruppe der Visualisierung identifiziert wurden: Die Nutzer der Visualisierung sollen sich grundlegend mit den Themen der Software-Entwicklung auskennen, jedoch kein detailliertes Wissen über die Umsetzung des betrachteten Software-Projekts haben.
Über das Probandensystem werden gezielt die Studierenden des Fachs Mensch-Computer-Systeme angesprochen. Um eine größere Gruppe an möglichen Teilnehmern zu erreichen, wird die Studie außerdem in der allgemeinen Versuchspersonenkartei aufgeführt.
Da es sich um eine explorative Studie handelt, ist keine bestimmte Anzahl von Teilnehmern notwendig, da keine statistisch signifikanten Werte erhoben werden müssen. Aufgrund der eingeschränkten Zeit zur Durchführung der Studie wurden 18 Termine angeboten. Insgesamt nahmen 12 Personen an der Studie teil.
Die Probanden wurden entweder durch das Anrechnen einer Versuchspersonenstunde oder mit 10€ für die Teilnahme an der Studie entschädigt.

6.2.2 Material

Hardware und Programme

Während der Nutzerstudie laufen die untersuchten VR-Anwendungen auf einem Computer mit folgenden Technischen Daten:

- Prozessor: Intel(R) Core(TM) i7-9700KF CPU 3.60GHz, 3600 MHz, 8 Kern(e), 8 logische Prozessoren
- RAM: 32.0 GByte
- Grafikkarte: NVIDIA GeForce RTX 2070 SUPER

Der Proband trägt eine HTC Vive Pro als Head-Mounted Display (HMD) und interagiert über zwei Controller mit der Visualisierung. Außerdem werden zwei Vive Basisstationen der zweiten Generation verwendet. Die Anbindung der VR-Hardware an den Computer erfolgt über SteamVR 1.9.16.
Der Spielbereich der VR-Anwendungen hat die Maße 3,5m x 2,3m.
Auf dem gleichen Computer wie die VR-Anwendungen läuft parallel die Anwendung Neo4J Desktop, die die notwendige Datenbank zur Verfügung stellt.
Die im folgenden beschriebenen Fragebögen werden vom Probanden in dem Online-Umfrage-Tool LimeSurvey ausgefüllt. Hierzu steht ein Laptop mit 15 Zoll Bildschirmdiagonale zur Verfügung.

Fragebögen

Zur Messung der subjektiven Nutzerzufriedenheit werden zwei standardisierte Fragebögen verwendet: Jede Aufgabe wird während der Usability-Studie direkt durch den After-Scenario Questionnaire (ASQ) (Lewis, 1991) bewertet. Die Zufriedenheit mit dem Gesamtsystem wird anschließend durch die System Usability Scale (SUS) (Brooke, 1995) bewertet. Um die kognitive Belastung des Probanden bei der Nutzung der Visualisierung zu messen, wird außerdem der NASA Task Load Index (NASA-TLX) (NASA, 1986) eingesetzt. Da es sich bei der Visualisierung um eine VR-Anwendung handelt, wird außerdem der Simulation Sickness Questionnaire (SSQ) (Kennedy, Lane, Berbaum & Lilienthal, 1993) genutzt, um sicherzustellen, dass die Anwendung keine negativen Auswirkungen auf das Wohlbefinden der Nutzer hat.

Weitere Werte

Quantitative Werte

Um die Effektivität zu bestimmen, mit der die Aufgaben gelöst wurden, wird die Richtigkeit der Antworten festgehalten. Dabei wird zwischen richtiger, falscher und unvollständiger Antwort unterschieden.
Um einen Anhaltspunkt für die Effizienz bei der Bearbeitung der Aufgaben zu erhalten, wird die Zeit gemessen, die der Proband dafür benötigt. Dabei startet die Messung der Bearbeitungszeit, wenn die Versuchsleiterin die Aufgabe vollständig vorgelesen hat. Die Messung endet, wenn der Proband alle Teilfragen der Aufgabe vollständig beantwortet hat, oder wenn er mitteilt, keine weiteren Antworten geben zu können.

Qualitative Werte

Da es sich um eine explorative Studie handelt, die auch zum Ziel hat, Aspekte der Visualisierung zu finden, die noch verbessert werden können, wird auch die qualitative Meinung des Nutzers abgefragt.
Dazu wird einerseits die Think-Aloud Methode angewandt. Der Proband wird dazu aufgefordert, während der Interaktion mit der Anwendung und der Bearbeitung der Aufgaben sein Vorgehen verbal zu beschreiben. Auffälligkeiten können ebenfalls sofort benannt werden.
Andererseits stehen dem Nutzer nach der Usability-Studie einige Freitextfelder zur Verfügung, um verschiedene Aspekte der Visualisierung und das Gesamtsystem zu bewerten. Die Fragen dazu sind:

- Was hat Ihnen beim Übergang zwischen zwei Zeitpunkten gefallen?
- Was hat Ihnen beim Übergang zwischen zwei Zeitpunkten nicht gefallen?
- Was hat Ihnen an der Darstellung der Inselwelt gefallen?
- Was hat Ihnen an der Darstellung der Inselwelt nicht gefallen?
- Was hat Ihnen an der Anwendung insgesamt gefallen?
- Was hat Ihnen an der Anwendung insgesamt nicht gefallen?
- Welche zusätzlichen Informationen oder Hilfen hätten Sie sich beim Lösen der Aufgabe gewünscht?
- Was möchten Sie noch zu der Anwendung sagen?

Demographische Daten

Im Rahmen der Studie werden folgende persönliche Daten des Probanden abgefragt. Wenn es sich bei dem Wert um eine Auswahl handelt, sind die Optionen in Klammern angegeben. Ansonsten handelt es sich um Zahl- oder Texteingaben.

- Geschlecht (weiblich, männlich, keine Antwort)
- Alter
- Studiengang
- Fachsemester
- höchster bisheriger Abschluss
  (Bachelor, Abitur/Fachabitur, Master, Promotion, anderes)
- Programmiererfahrung (Anfänger, Fortgeschritten, Experte)
- Anlass der Programmiererfarung
  (Studium/Schule, private Projekte, Beruf, Sonstiges)
- Erfahrung mit VR-Anwendungen (Ja, Nein)

6.2.3 Beschreibung der untersuchten Systeme

In der Nutzerstudie werden drei Darstellungsvarianten untersucht. Diese basieren alle auf der in Kapitel 5.1.1 beschriebenen Inselmetapher. Jedoch weisen sie unterschiedlich starke Optimierungen im Hinblick auf die Darstellung von Software-Historie auf.
Alle Systeme beinhalten die in Kapitel 5.2 beschriebenen Pfeiltasten zur Navigation zwischen dem momentan angezeigten und dem nachfolgenden bzw. vorherigen Commit. Außerdem wird in allen Varianten innerhalb der Visualisierung angezeigt, welcher Commit gerade dargestellt wird.
Die Varianten werden im Folgenden als System a), System b) und System c) bezeichnet:

System a)

Bei System a) handelt es sich um die Darstellung der Software-Architektur durch die ursprüngliche Anwendung IslandViz (Kapitel 5.1.1). Um diese Anwendung mit den anderen Systemen vergleichen zu können, wurde sie um die Elemente zur Navigation innerhalb der Zeit erweitert. Um einen neuen Commit anzuzeigen, lädt die

Anwendung eine neue Unity-Scene, in der die Inselwelt auf dem Tisch dem gewählten Zeitpunkt entspricht. Bis der Ladevorgang der neuen Scene abgeschlossen ist, wird im Blickfeld des Nutzers eine hellgraue Fläche eingeblendet.

System b)

Bei System b) handelt es sich um die, im Rahmen dieser Arbeit konzeptionierte und implementierte, Erweiterung von IslandViz. Die Anzeige eines neuen Commits erfolgt, indem die Inseln auf dem Tisch an ihre neue Position verschoben und die Mesh-Komponenten der Inseln erneuert werden. Die Hervorhebung von Änderungen ist deaktiviert.

System c)

System c) entspricht System b). Hier ist zusätzlich die Hervorhebung von Änderungen gemäß Kapitel 4.3.4 aktiviert.

Ein Vergleich der untersuchten Systeme befindet sich in Anhang D.

6.2.4 Prozedur

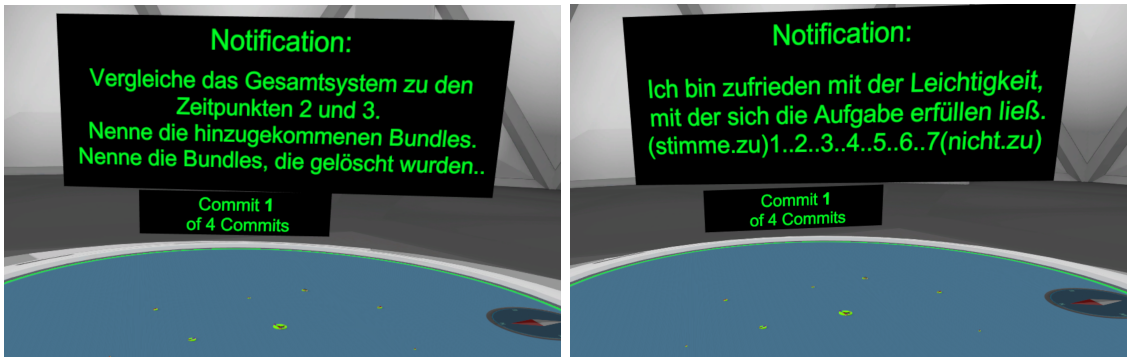
Zuordnung der Probanden

Die Probanden werden zufällig in zwei gleich große Gruppen aufgeteilt. Die Probanden der ersten Gruppe testen nur System a). Die Probanden der zweiten Gruppe testen System b) und System c).

Ablauf

Der Ablauf der Test-Sessions wird für beide Gruppen in Tabelle 6.1 zusammengefasst.
Alle Probanden müssen zunächst den Hinweisen zum Datenschutz zustimmen und füllen einen Fragebogen zur Erhebung der demographischen Daten aus (vgl. Kapitel 6.2.2). Anschließend werden die Probanden gebeten, den SSQ auszufüllen.
Um den Probanden mit der Visualisierung vertraut zu machen, wird ihm ein Text zur Einführung in die dargestellte Software-Architektur, die verwendete Inselmetapher und die Navigation innerhalb der Anwendung vorgelegt und der Use-Case

beschrieben (siehe Anhang C).
Danach kann der Proband das HMD aufsetzen und sich mit der Visualisierung vertraut machen. Währenddessen kann der Proband Fragen stellen und die Versuchsleiterin erläutert alle wichtigen Aspekte der Darstellung.
Zur Durchführung der eigentlichen Usability-Studie werden dem Probanden zwei Aufgaben pro System gestellt (siehe Kapitel 6.2.4). Jede Aufgabe kann in Textform innerhalb der VR-Anwendung auf Wunsch des Proband ein- und ausgeblendet werden (Abbildung 6.1a). Die Versuchsleiterin liest die Frage zu Beginn jeweils einmal vor, woraufhin die Zeitmessung beginnt.
Während der Proband die Aufgabe löst, notierte sich die Versuchsleiterin handschriftlich die in Kapitel 6.2.2 beschriebenen weiteren Werte.
Im Anschluss an jede Aufgabe beantwortet der Proband die Fragen des ASQ. Dazu werden die Fragen in die VR-Umgebung eingeblendet (Abbildung 6.1b) und die Bewertung des Probanden von der Versuchsleiterin in das Umfrage-Tool eingegeben.
Nach der Bearbeitung beider Aufgaben füllt der Proband im Umfrage-Tool die SUS, den NASA-TLX, die Freitextfragen zum System, und den SSQ aus.
Die Probanden der zweiten Gruppe wiederholen den Ablauf anschließend mit System c).
Als letzter Schritt folgt für alle Probanden die Gewichtung der Dimensionen des NASA-TLX.



(a) Aufgabenstellung **(b)** Bewertungsaussagen des ASQ

Abbildung 6.1. Texteinblendungen in der VR-Anwendung für den Studienverlauf

Tabelle 6.1. Studienablauf

Gruppe 1	Gruppe 2
Datenschutzerklärung demographische Daten SSQ Einführungstexte	Datenschutzerklärung demographische Daten SSQ Einführungstexte
Vertrautmachen mit der Anwendung	Vertrautmachen mit der Anwendung
Aufgabe 1 in System a) ASQ Aufgabe 2 in System a) ASQ SUS NASA-TLX Freitextfragen SSQ	Aufgabe 1 in System b) ASQ Aufgabe 2 in System b) ASQ SUS NASA-TLX Freitextfragen SSQ
	Aufgabe 3 in System c) ASQ Aufgabe 4 in System c) ASQ SUS NASA-TLX Freitextfragen SSQ
NASA-TLX Gewichtung	NASA-TLX Gewichtung

Aufgabenbeschreibung

Die Aufgaben, die die Probanden in der Studie lösen sollen, sind so gestaltet, dass in jedem System zunächst die Veränderungen auf Ebene des gesamten Software-Systems betrachtet werden. Anschließend sollen die Änderungen innerhalb eines Bundles benannt werden.
Die Aufgaben 1 und 2 werden von Gruppe 1 in System a) und von Gruppe 2 in System b) bearbeitet. Die Aufgaben 3 und 4 werden nur von Gruppe 2 in System c) bearbeitet. Diese Aufteilung ermöglicht einen direkten Vergleich zwischen den Systemen a) und b). Für die Probanden der zweiten Gruppe sind zusätzliche Aufgaben für System c) notwendig, um zu verhindern, dass sich Probanden an die Antworten erinnern. Die Aufgaben 1 und 3 und die Aufgaben 2 und 4 sind dabei so gestaltet, dass die Bedingungen wie Anzahl der angezeigten Inseln, Regionen und Gebäude,

sowie die Komplexität der erwarteten Antwort vergleichbar sind.
Die Tabellen 6.2 bis 6.5 fassen die Aufgaben mit ihren Unterpunkten und den erwarteten Antworten zusammen.

Tabelle 6.2. Aufgabe 1
Vergleiche das Gesamtsystem zu den Zeitpunkten 2 und 3.

Nenne die hinzugekommenen Bundles.	GUI Command, Data Model
Nenne die Bundles, die gelöscht wurden.	Scripting

Tabelle 6.3. Aufgabe 2
Vergleiche das Bundle Data Model an den Zeitpunkten 3 und 4.

Welche Packages wurden hinzugefügt?	TestUtils
Welche Packages wurden gelöscht?	kein Package wurde gelöscht
Welche Dateien wurden in bestehenden Paketen hinzugefügt?	DefaultTypedDatumConverter, DefaultTypedDatumFactory, DefaultTypedDatumSerializer
Welche Dateien wurden gelöscht, deren Packages noch existieren?	TypedDatumServiceImpl

Tabelle 6.4. Aufgabe 3
Vergleiche das Gesamtsystem zu den Zeitpunkten 3 und 4.

Nenne die hinzugekommenen Bundles.	GUI Login, Input Provider
Nenne die Bundles, die gelöscht wurden.	Shutdown

Tabelle 6.5. Aufgabe 4
Vergleiche das Bundle Executor an den Zeitpunkten 3 und 4.

Welche Packages wurden hinzugefügt?	FileInfo
Welche Packages wurden gelöscht?	SPI
Welche Dateien wurden in bestehenden Paketen hinzugefügt?	LocalCommandLineExecutor, AbstractCommandLineExecutor, ProcessExtractor
Welche Dateien wurden gelöscht, deren Packages noch existieren?	RCECommandLineExecutor

6.3 Ergebnisse

6.3.1 Durchführung

Die Studie fand vom 18. bis 21. Februar 2020 im Mint-Lab des Lehrstuhls für Human-Computer Interaction an der Universität Würzburg statt. Die Durchführung der Studie dauerte bei Probanden der Gruppe 1 im Durchschnitt 50 Minuten. Probanden der Gruppe 2 benötigten im Durchschnitt 62 Minuten. Die etwas längere Gesamtzeit bei Gruppe 2 war erwartet worden, da diese Probanden zwei Systeme beurteilen sollten. Die Bearbeitungszeit der einzelnen Aufgaben war bei dieser Gruppe jedoch kürzer, sodass sich eine zeitliche Differenz von nur 12 Minuten ergibt.

6.3.2 Soziodemographische Daten

Die soziodemographischen Eigenschaften aller Probanden der Studie sind in Abbildung 6.2 zusammengefasst. Von insgesamt 12 Teilnehmern waren 7 männlich und 5 weiblich. Die Hälfte der Teilnehmer ließ sich der Altersgruppe 19 bis 22 Jahre zuordnen. Die übrigen Probanden verteilten sich gleichmäßig auf die Altersgruppen 23 bis 26 Jahre, 30 bis 33 Jahre und älter als 33 Jahre, wobei der älteste Proband 52 Jahre alt war. Ein Viertel der Probanden befand sich im ersten Fachsemester seines Studiums, ein weiteres Viertel im siebten Semester. Ein Proband hatte in dieser Kategorie keine Angabe gemacht. Die Mehrheit der Probanden (10 von 12) gab als höchsten Bildungsabschluss das Abitur oder Fachabitur an. Je ein Proband hatte einen Bachelor- bzw. Masterabschluss. Ein Drittel der Probanden war Student des Fachs Mensch-Computer-Systeme. Ein weiteres Drittel studierte ein Fach mit Informatikanteilen. Das letzte Drittel der Probanden hatte andere fachliche Hintergründe.
Die Programmiererfahrung der Probanden wurde ebenfalls abgefragt. Gemäß Abbildung 6.3 bezeichnet sich jeweils die Hälfte der Probanden als Anfänger bzw. als fortgeschritten im Programmieren. Die Programmiererfahrung stammt dabei bei drei Viertel der Probanden aus der Schule oder dem Studium. Weitere fünf gaben eigene Projekte an und ein Proband hat Erfahrung aus dem beruflichen Kontext. Mehrfachnennungen waren in diesem Fall möglich.
Elf von zwölft Probanden hatten bereits als Nutzer Erfahrung mit VR-Anwendungen gemacht.

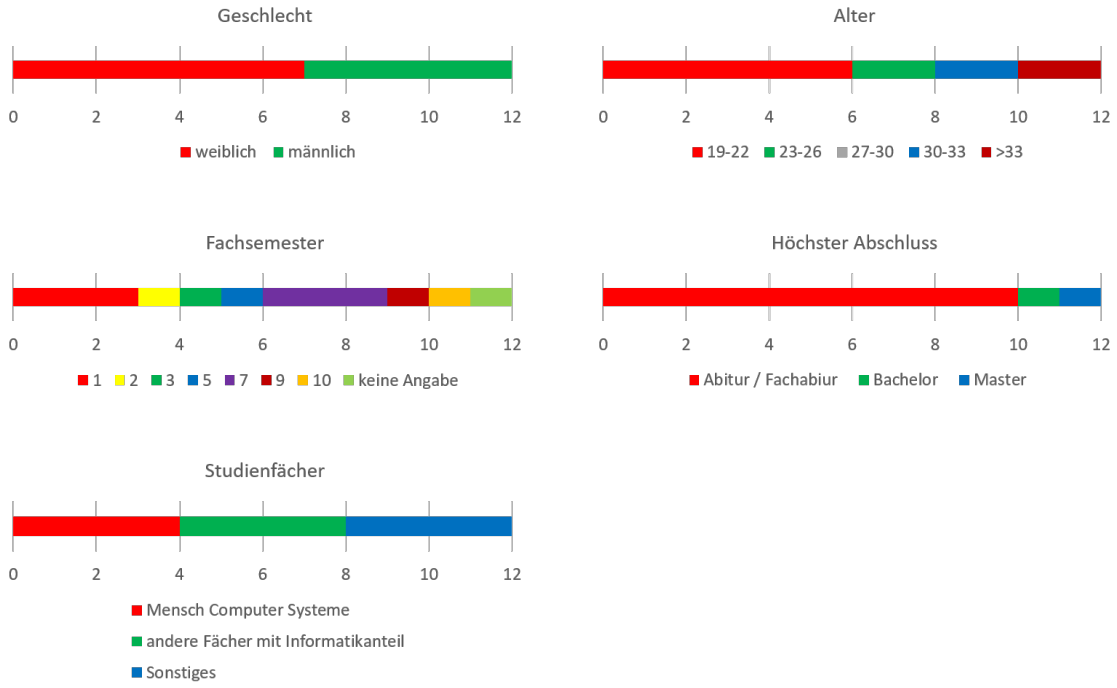


Abbildung 6.2. Zusammenfassung der soziodemographischen Daten der Probanden

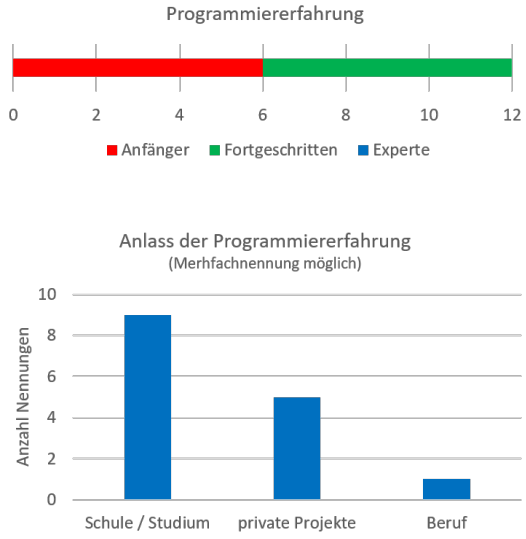


Abbildung 6.3. Programmiererfahrung der Probanden

6.3.3 Richtigkeit der Antworten

Bewertung

Bei den Aufgaben auf System-Ebene wurden gemäß der Tabellen 6.2 und 6.4 je drei Nennungen für eine vollständig korrekte Antwort erwartet. Es konnten in diesem Aufgabenbereich somit ingesamt drei Punkte erreicht werden. Für jede erwartete Antwort, die nicht genannt wurde, wurde ein Punkt abgezogen. Falsche Nennungen sind nicht berücksichtigt.
Analog konnten bei den Aufgaben auf System-Ebene insgesamt sechs Punkte (vgl. Tabellen 6.3 und 6.5) erreicht werden.

Bei gemeinsamer Betrachtung beider Aufgaben, konnten die Probanden in System b) (M=98.15%, SD=4.54%) bessere Werte erzielen als in System c) (M=85.19%, SD=11.48%) und System a) (M=74.07%, SD=30.36%).
Abbildung 6.4 zeigt die Ergebnisse im Detail nach Aufgabe getrennt.
Mit System b) konnten alle Probanend die Aufgabe auf Bundle-Ebene vollständig erfolgreich lösen. Das System c) ergab bei beiden Aufgaben schlechtere Ergebnisse als System b). System a) schnitt bei beiden Aufgaben im Durchschnitt am schlechtesten ab.

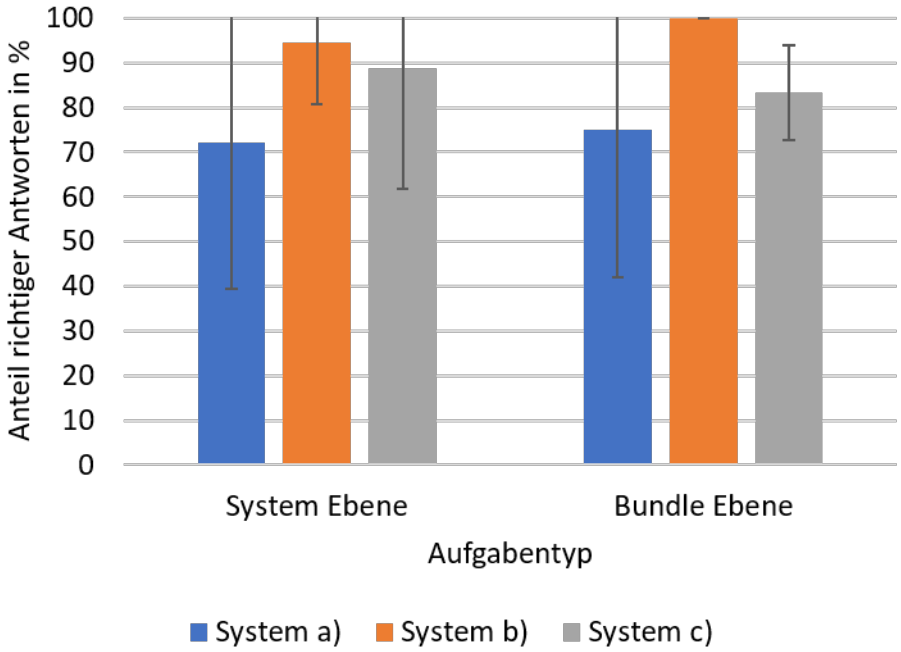


Abbildung 6.4. Mittelwert und Standardabweichungen des Anteils der richtigen Antworten in % nach Aufgabentyp und System

6.3.4 Bearbeitungszeit

Die Bearbeitungszeit für beide Aufgaben und alle drei Systeme ist in Abbildung 6.5 dargestellt.
Für die Aufgaben auf System-Ebene ergeben sich für Systeme a) (M=234.17 s, SD=77.62 s) und System b) (M=236.00 s, SD=105.07 s) ähnliche Werte. Die Proganden lösten die Aufgabe in System c) deutlich schneller (M=59.5 s, SD=36.09 s). Bei den Aufgaben auf Bundle-Ebene zeigt sich eine deutlicher Unterschied zwischen den Systemen.

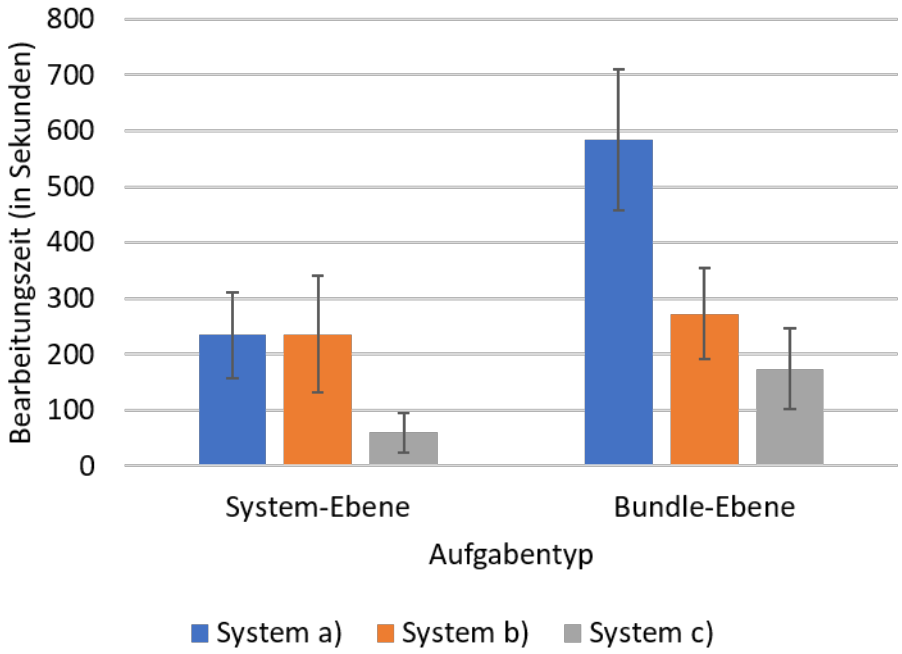


Abbildung 6.5. Mittelwert und Standardabweichung der Bearbeitungszeit in Sekunden nach Aufgabentyp und System

6.3.5 After Scenario Questionnaire

Bewertung

Beim ASQ werden nach jeder Aufgabe drei Aussagen auf einer siebenstufigen Likert-Skala bewertet. Dabei entspricht der Wert 1 “Ich stimme voll zu.” und der Wert 7 “Ich stimme gar nicht zu.” (Lewis, 1991). Somit liegt bei jeder Aufgabe die beste Bewertung bei 3 und die schlechteste bei 21. Entsprechend können die Bewertungen der Gesamtsysteme nach zwei Aufgaben zwischen 6 und 42 liegen.

Die Werte der Gesamtsysteme zeigen, dass die Systeme b) und c) besser besser bewertet werden als System a) (M=20.50, SD=4.64). Wobei System c) (M=12, SD=3.90) bessere Bewertungen erhält als System b) (M=16.67, SD=5.34).
Die Bewertungen nach Aufgabe aufgeschlüsselt werden in Abbildung 6.6 dargestellt. Es fällt auf, dass die Bewertungen nach dem ASQ scheinbar mit der Bearbeitungszeit der Aufgaben korrelieren. Die Zufriedenheit mit den Systemen a) und b) unterscheidet sich bei der Aufgabe auf System-Ebene kaum. System c) wird in diesem Fall besser bewertet. Bei den Aufgaben auf Bundle-Ebene wird System a) deutlich schlechter bewertet als die Systeme b) und c).

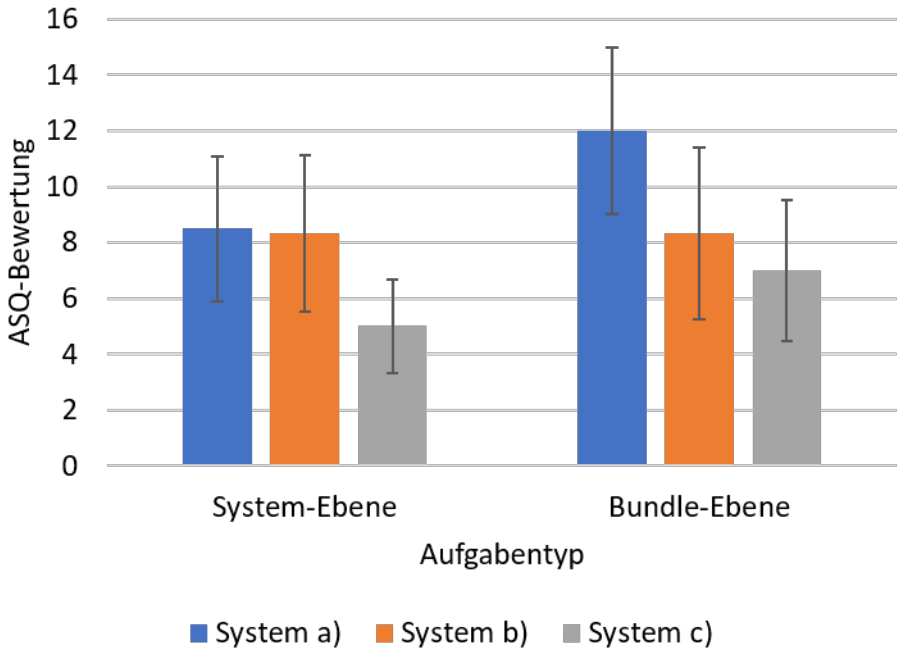


Abbildung 6.6. Mittelwerte und Standardabweichungen der ASQ-Bewertung nach Aufgabentyp und System
(bester möglicher Wert: 3, schlechtester möglicher Wert: 21)

6.3.6 System Usability Scale

Bewertung

Bei der SUS werden 10 Aussagen auf einer fünfstufigen Skala von 1 (“Stimme überhaupt nicht zu”) bis 5 (“Stimme voll zu”) bewertet. Die Hälfte der Aussagen sind positiv formuliert und gehen mit Wert-1 in die Bewertung ein. Die andere Hälfte enthält eine Negation und wird mit 5-Wert in die Bewertung eingezogen. Die Summe der berechneten Einzelwertungen der Aussagen wird mit 2.5 multipliziert (Brooke, 1995). Damit ergibt sich eine Bewertung zwischen einem schlechtesten Wert von 0 und einem besten Wert von 100.

Wie Abbildung 6.7 zeigt, ergibt sich für die Systeme nur eine geringfügig unterschiedliche Wertung: System a) (M=72.92, SD=10.66), System b) (M=73.75, SD=18.29), System c) (M=80.42, SD=19.90).
Vergleicht man die erreichten Werte mit der von Bangor, Kortum und Miller (2009) gegebenen Tabelle zur Einordnung von SUS-Ergebnissen, können alle drei Systeme als annehmbar bezeichnet werden. System a) und System b) erreichen demnach eine Note ’C’ und System c) die Note ’B’. (Wobei ’A’ der besten Note entspricht).

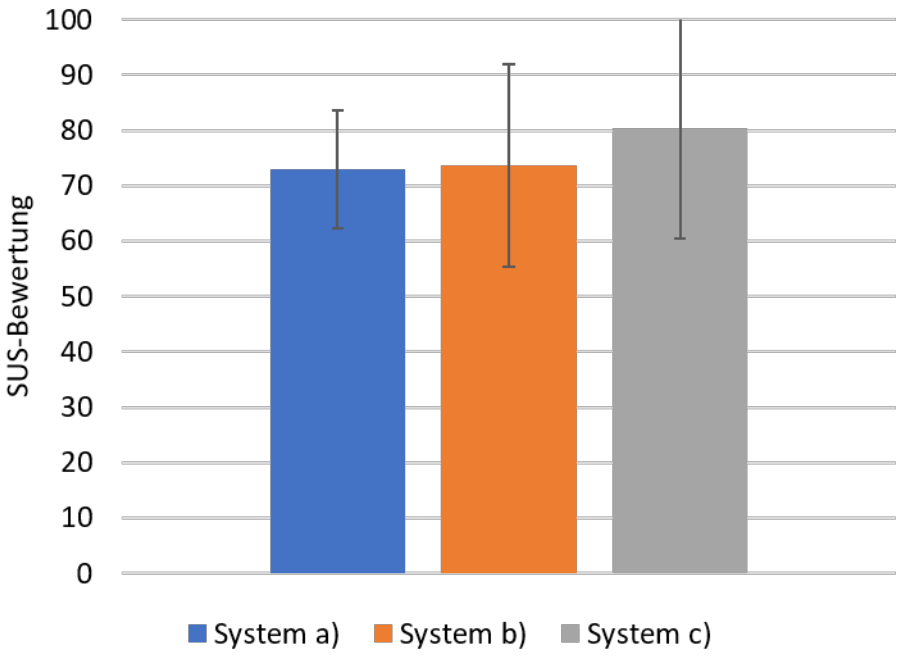


Abbildung 6.7. Mittelwerte und Standardabweichung der SUS-Bewertung der Gesamtsysteme
(bester möglicher Wert: 100)

6.3.7 NASA Task Load Index

Bewertung

Beim NASA-TLX wird der Workload der beim Bearbeiten von Aufgaben im untersuchten System besteht durch sechs Dimensionen ermittelt. Der Proband gibt dazu für jede Dimension an, wie stark die Belastung war. Dazu steht eine Skala von 0 bis 100 mit einer Schrittweite von 5 zur Verfügung. Dabei entspricht 0 einer geringen und 100 einer starken Belastung. Zusätzlich bewertet der Proband, welche der Dimensionen für den Workload besonders entscheidend sind. Hierzu legt er für alle 15 Paarungen der Dimensionen fest, welche der beiden er jeweils als entscheidender betrachtet. Dabei kann eine Dimension einen Maximalwert von 5 erreichen. Hieraus ergibt sich für jede Dimension ein Faktor, mit dem der jeweilige Wert multipliziert wird. Die Summe aller Dimensionen wird durch Division durch 15 auf einen Wertebereich zwischen 0 und 100 normiert und ergibt einen Gesamtwert (NASA, 1986).

Der normierte Gesamtwert des NASA-TLX ergab für System a) (M=47.44, SD=16.52) eine mittlere Belastung. System b) erhält eine ähnliche Bewertung (M=40.50, SD=22.96). Die Belastung bei System c) wird von den Probanden geringer eingestuft (M=20.50, SD=16.49).
Abbildung 6.8 zeigt die Gewichtungen der Belastungsdimensionen nach Probandengruppe. In den meisten Fällen werden die Dimensionen ähnlich bewertet. Die Geistige Anforderung wird in beiden Gruppen als hauptsächlicher Faktor der Belastung betrachtet. Es fällt jedoch auf, dass Gruppe 1, die System a) getestet hat, den Faktor Frustration deutlich stärker gewichtet als Gruppe 2.
Die gewichteten Bewertungen aufgeteilt nach Dimension und untersuchtem System werden in Abbildung 6.9 dargestellt.

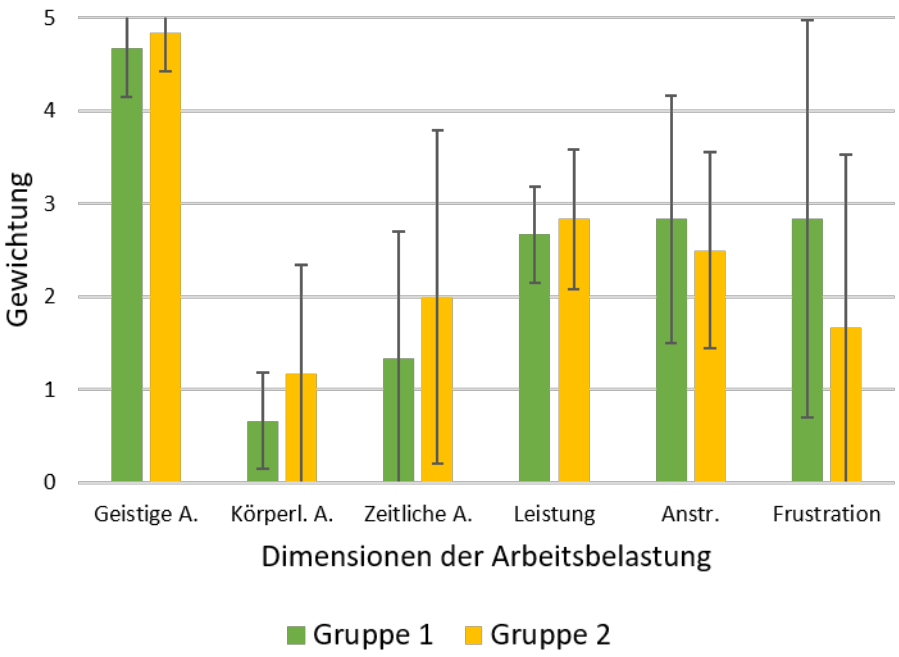


Abbildung 6.8. Mittelwert und Standardabweichung der Gewichtung der einzelnen Dimensionen des NASA-TLX nach Probanden-Gruppe

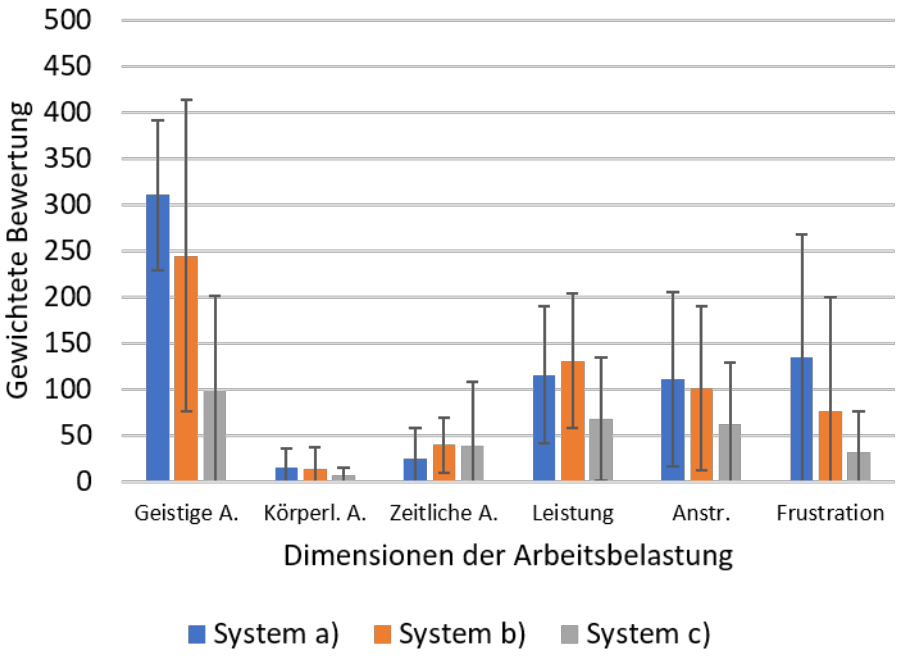


Abbildung 6.9. Mittelwert und Standardabweichung der gewichteten Bewertung der einzelnen Dimensionen des NASA-TLX nach Aufgabentypen

6.3.8 Simulator Sickness Questionnaire

Bewertung

Für den SSQ werden eine Reihe von Symptomen abgefragt, für die der Proband auf einer vierstufigen Scala von 0 (“keine”) bis 3 (“stark”) angibt, wie er diese momentan bei sich wahrnimmt. Die Symptome sind nach Kennedy et al. (1993) drei Symptom-Gruppen zugeordnet. Diese sind Augenbewegung (O, engl.: Oculomotor), Desorientierung (D, engl.: Disorientation) und Übelkeit (N, engl.: Nausea). Um die Werte der Gruppen zu bestimmen, werden die Bewertungen der einzelnen, zugehörigen Symptome addiert und die Summe mit spezifischen Faktoren multipliziert. Ein Gesamtwert (TS) ergibt sich durch die Addierten Werte der Symptom-Gruppen ohne die Einbeziehung der Faktoren. Nur die Gruppe D geht mit einem weiteren Faktor ein.
Der SSQ wurde jeweils einmal vor und nach der Nutzung des Systems ausgefüllt um die Differenz der Werte zu erhalten.

Die SSQ-Bewertungen nach der Nutzung der Systeme ist in Abbildung 6.10 zu sehen. Es handelt sich bei allen Kategorien um sehr kleine Werte, da die Probanden in den meisten Fällen angaben keine oder kaum Symptome zu spüren. Abbildung 6.11 zeigt außerdem wie sich die SSQ-Bewertungen nach der Nutzung der Systeme verändert haben. Bei negativen Werten spührten die Probanden nach der Nutzung weniger Symptome als zuvor.
Aufgrund technischer Probleme konnten die SSQ-Bewertungen bei System a) nur von zwei der sechs Probanden ausgewertet werden.

6.3.9 Qualitative Bewertungen

Die qualitativen Bewertungen der Nutzer in Form von Antworten auf Freitextfragen und Aussagen während der Nutzung der Anwendung sind vollständig in Anhang E aufgeführt.

Bei den Bewertungen können drei häufig genannte Themenfelder identifiziert werden. Bei diesen Handelt es sich um den Erhalt der Mentalen Karte in den Systemen b) und c) bzw. die Neupositionierung der Elemente in System a). Außerdem wurden einige Anmerkungen zur Umsetzung der farblichen Hervorhebungen in System c) und zum Umgang mit gelöschten Elementen in den Systemen b) und c) gemacht.

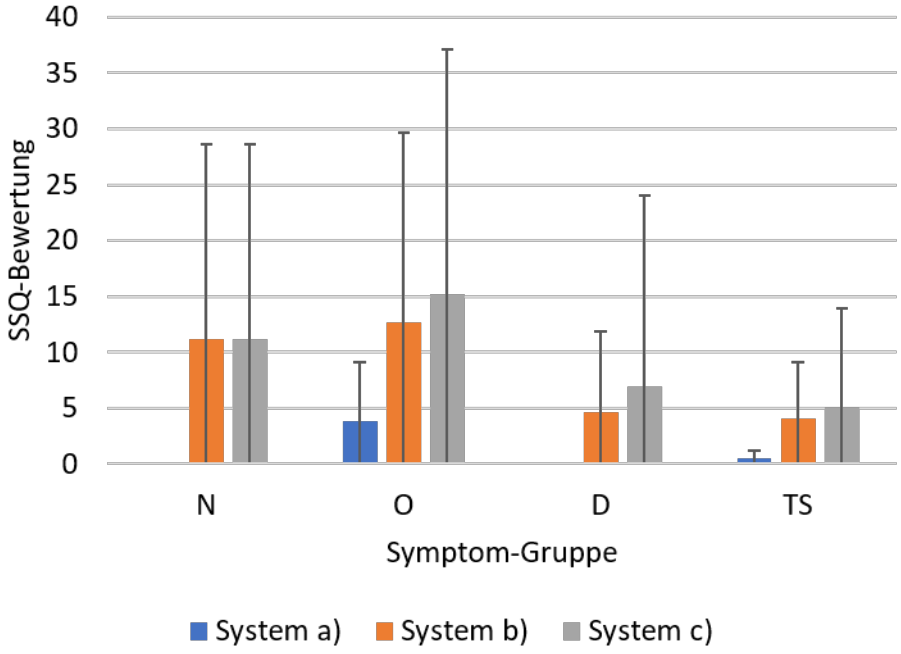


Abbildung 6.10. Ergebnisse des SSQ nach der Benutzung der Systeme

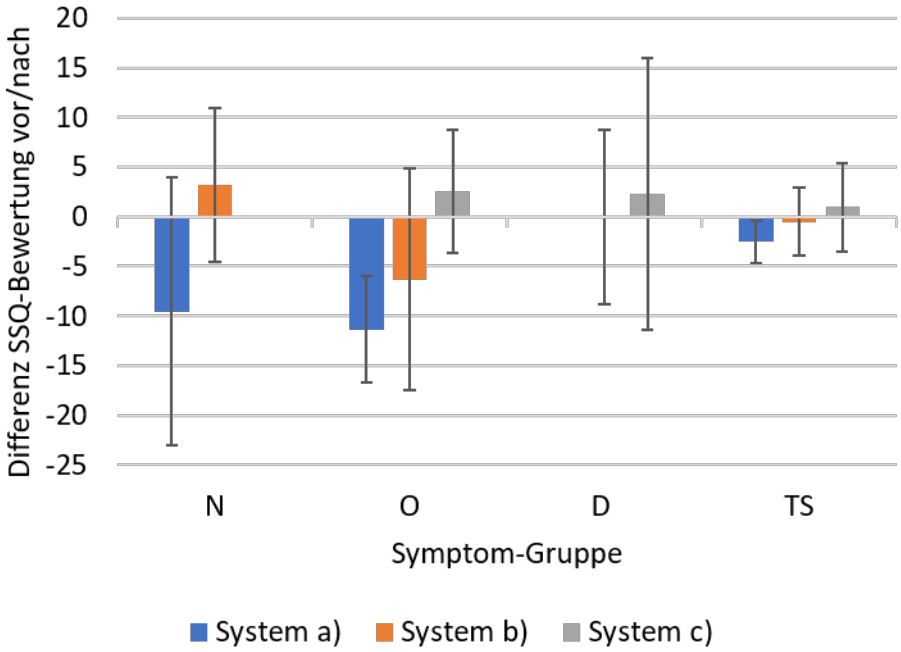


Abbildung 6.11. Differenz der Ergebnisse des SSQ vor und nach der Benutzung der Systeme

Die zu diesen Themenfeldern gehörenden Aussagen fließen in den folgenden Abschnitt ein, um die Schlussfolgerungen zu unterstützen.

Im Folgenden werden noch einige Punkte zusammengefasst, die jeweils nur von ein bis zwei Probanden bei System b) oder System c) als Störend empfunden wurden. Die Kritikpunkte an System a) werden hier nicht weiter erwähnt, da dieses System nicht Schwerpunkt der Arbeit ist und lediglich als Referenz diente.
Beim Übergang zwischen Commits wurde bemängelt, dass das eingeblendete Informationsfeld (vgl. Abbildung 5.6b) die Sicht auf die Inseldarstellung störte. Weiterhin wurde die Zeit, die für den Übergang benötigt wird, als zu lang empfunden. Einigen Probanden fiel außerdem auf, dass die Namensschilder neuer Inseln nicht automatisch eingeblendet werden und sich die Informationstafel nicht aktualisiert.
Ein Problem innerhalb der Implementierung besteht, wenn die Inseln an ihre neue Position verschoben werden. Wenn der Proband zu diesem Zeitpunkt stark an eine Insel herangezoomt hatte, schwamm diese beim Übergang zwischen Commits aus dem sichtbaren Bereich (*"Ist das normal, dass die Insel so wegschwimmt?"*). Ein Proband hatte außerdem Schwierigkeiten alle Gebäude einer Insel zu überblicken, da kleinere Gebäude durch größere Nachbarn oder aufgrund der Topologie der Insel verdeckt wurden.

6.4 Diskussion

Die Nutzerstudie sollte zeigen, dass das im Rahmen dieser Arbeit entworfene Konzept geeignet ist, Änderungen zu identifizieren, die an einem Software-Projekt gemacht wurden. Als sekundäres Ziel sollte außerdem untersucht werden, ob eine farbliche Hervorhebung der Änderungen dem Nutzern die Bearbeitung der Aufgaben erleichtert. Außerdem sollten durch die qualitativen Aussagen der Nutzer Verbesserungsmöglichkeiten für das Konzept gefunden werden.

6.4.1 Zusammenfassung der Ergebnisse

Bei den Befragungen zur Nutzerzufriedenheit konnten alle drei Systeme zufriedenstellende oder gute Werte erzielen. Im direkten vergleich schnitt System a) jedoch am schlechtesten ab und System c) erreichte etwas bessere Werte als System b).

Im Hinblick auf Richtigkeit und Bearbeitungszeit zeigten sich deutliche Unterschiede. Die Probanden konnten bei System b) im Durchschnitt 33% mehr richtige Antworten geben als bei System a). In system b) wurden außerdem insgesamt 98% der möglichen Punkte erzielten. Im System c) wurden durchschnittlich etwa 1.4 Punkte weniger erreicht als in System b). Dies entspricht Differenz von 15%. Bei der Bearbeitungszeit benötigten die Probanden für die Aufgabe auf System-Ebene in System a) und b) etwa gleich lange und in System c) nur ein Drittel der Zeit. Für die Aufgaben auf Bundle-Ebene wurde in System b) die Hälfte der Zeit benötigt wie in System a). Die Bearbeitungszeit in System c) betrug wiederum nur zwei Drittel der Zeit in System b).
Die wahrgenommene Belastungen ist in System c) am geringsten und in System a) am stärksten. Der Vergleich der mentalen Anstrengung zeigt, dass diese in System b) um 21% geringer wahrgenommen wird als in System a). Außerdem scheint in System a) der Faktor Frustration eine größere Rolle zu spielen.

6.4.2 Schlussfolgerungen

Trotz der ähnlichen Bewertungen der Systeme im Hinblick auf die Nutzerzufriedenheit lässt sich feststellen, dass die konzeptionierte Erweiterung besser geeignet ist, Software-Historie darzustellen, als eine Visualisierung, die auf statische Software-Architektur optimiert ist.
Anhand der Aussagen der Nutzer lässt sich vermuten, dass die korrektere und schnellere Bearbeitung sowie die geringere Belastung durch den Erhalt der mentalen Karte erklärt werden kann, der Schwerpunkt dieser Arbeit war. Nutzer von System a) kritisierten: *“Änderungen nicht im System wahrnehmbar, muss aus dem Gedächtnis verglichen werden” “Die Positionen der Inseln waren nicht über die Zeitpunkte hinweg an derselben Stelle.”* Im Vergleich dazu lobten Nutzer des Systems b): *“Man sieht sofort die Unterschiede” “Der Bildausschnitt ist ungefähr gleichgeblieben, sodass man gut vergleichen konnte”.*

Die sekundäre Fragestellung untersuchte, ob ein positiver Effekt durch farbliche Hervorhebungen auftritt. Tatsächlich nannten die Probanden sowohl bei System a) als auch bei System b) in den Fragen nach negativen Aspekten, dass Änderungen nicht markiert wurden: *“Keine Markierung oder Hervorhebung der Veränderungen”*, *“Zu wenig Infos über Änderungen zwischen 2 Commits”*. Dieser Aspekt wurde ebenfalls

in der Frage nach zusätzlich gewünschten Informationen aufgegriffen. An dieser Stelle wurden auch verschiedene Vorschläge zur Umsetzung gemacht: *"Eine Anzeige auf der Tafel, welche Bausteine sich verändert haben (gelöscht, hinzugefügt)." "Farbliche Hervorhebung der Veränderungen"*
Entsprechend bewerteten die Probanden die farblichen Hervorhebungen in System c) positiv: *"Die grünen Ringe haben Veränderungen sehr gut dargestellt. Im Prinzip das, was ich mich bei der vorherigen Anwendung (ohne Ringe) gewünscht hätte"*, *"Hervorhebung der Veränderungen sehr gut deutlich durch auffällige/leuchtende Farben"*

Es zeigte sich jedoch, dass die Hervorhebung von Änderungen noch verbessert werden muss. Auf System-Ebene besteht das Problem, dass die Größe der farbigen Markierung proportional zur Größe der Insel ist. Dadurch fielen die Änderungen an kleinen Inseln in der Gesamtansicht kaum auf und wurden von einigen Probanden übersehen: *"Bei sehr kleinen Inseln waren grüne Ringe teils sehr unscheinbar"*
Das Konzept sieht bisher nur eine Hervorhebung der geänderten Inseln und Gebäude vor. Die Änderung der Regionen sollte anhand der Änderung der Gebäude ableitbar sein. Hier forderten die Nutzer eine zusätzliche Hervorhebung von Änderungen an den Regionen: *''Man hat nicht auf den ersten Blick gesehen, welche Pakete dazugekommen sind bzw. sich in der Größe verändert haben."* Diese beiden Aspekte erklären auch das schlechtere Abschneiden des Systems c) bei der Richtigkeit der Antworten gegenüber System b). Die Probanden verließen sich auf die Hervorhebung der Inseln und übersahen dabei eine kleinere Insel, die hinzugefügt wurde. Außerdem wurden Änderungen am Package oft falsch interpretiert.

Der größte weitere genannte Kritikpunkt am Konzept war der Umgang mit gelöschten Elementen. Diese werden nicht mehr dargestellt. Probanden meinten dazu: *"Wenn in der aktuell angezeigten Version ein Bundle oder ein Package nicht mehr vorhanden ist, wäre es einfacher zu erkennen, wenn dieser Bestandteil noch halbdurchsichtig o.ä. angezeigt würde."* Und empfanden als negativ, *"Dass im neuen Zeitpunkt nicht mehr vorhandene Dinge nicht mehr angezeigt werden."*

Einige Probanden erkannten außerdem den Grundgedanken der Visualisierung: *"Es wirkt ein bisschen wie Arbeit mit Gamification. Es macht Spaß mit dem Programm umzugehen."*, *"dadurch Bezug zur Realität und intuitives Verständnis der Welt"*

6.4.3 Einschränkungen

Wahl der Probanden

Bei der eigentlichen Zielgruppe der Anwendung handelt es sich um Projektleiter, die den Arbeitsumfang von Aufgaben bewerten sollen. Da es im Rahmen dieser Arbeit nicht möglich war, entsprechende Probanden zu rekrutieren, wurden stattdessen Studenten mit Informatik-Hintergrund gewählt. Diese Probanden entsprechen der tatsächlichen Zielgruppe insofern, dass sie zumindest grundlegende Kenntnisse im Bereich Software-Entwicklung haben und nicht mit der konkreten und detaillierten Implementierung des Software-Projekts vertraut sind. In der Realität ist jedoch anzunehmen, dass Projektleiter eigene Hintergrundinformationen zu dem betrachteten Software-Projekt haben. Dabei könnte es sich um bekannte Projektabschnitt, Wissen über Verbesserungsiterationen o.ä. handeln. Daher könnte zukünftig untersucht werden, wie diese Kenntnisse die Arbeit mit der Visualisierung beeinflussen.
Außerdem sollte betrachtet werden, ob Projektleiter die Visualisierung allein nutzen, oder ob sie gemeinsam mit anderen Personen betrachtet wird, z.B. im Rahmen von Planungsrunden oder Präsentationen. In diesem Fall würden mehrere Mitarbeiter ihr jeweiliges zusätzliches Wissen mit einbringen, was wiederum die Nutzung der Visualisierung verändern könnte.

Verwendeter Datensatz

Bei dem in der Studie visualisierten Datensatz handelte es sich um einen kleinen Ausschnitt der Architektur des Software-Projekts RCE. Dieses besteht insgesamt aus 163 Bundles. Für die Evaluation wurden lediglich 11 bis 13 Bundles verwendet. Damit kann die Studie nicht beantworten, ob die Erweiterung der Visualisierung zur Darstellung von Software-Historie auch für reale, große Software-Projekte nutzbar ist. Die Reduktion der dargestellten Bundles war jedoch notwendig, um einen Vergleich mit der ursprünglichen Anwendung zu ermöglichen, da es den Probanden in dieser bereits bei der sehr kleinen Anzahl von Bundles schwer fiel, Änderungen zu benennen.

Methodik

Das Design zum Vergleich der Systeme a) und b) entsprach einer Between-Studie, bei der Probanden der Gruppe 1 das System a) und Probanden der Gruppe 2 das

System b) bewerteten. Zur Beantwortung der sekundären Frage nach der Auswirkung zusätzlicher Highlights war es notwendig, dass die Probanden von Gruppe 2 zusätzlich das System c) testen. Um einen validen Vergleich der Systeme a) und b) zu gewährleisten, wurde der Ablauf für Gruppe 2 so geplant, dass immer zuerst System b) und anschließend System c) erprobt wurden.
Die so ermittelten Ergebnisse für System c) sind damit in sofern verfälscht, dass die Probanden durch die vorherige Nutzung von System b) bereits stärkere Vorerfahrung mit der Navigation im System und den Grundzügen der Visualisierung hatten. Dies könnte sich in den Bearbeitungszeiten der Aufgabe ausgewirkt haben.

6.4.4 Weiterführende Fragestellungen

Entsprechend der oben genannten Beschränkung in Bezug auf den untersuchten Datensatz sollte im nächsten Schritt mit einem veränderten Studienaufbau untersucht werden, ob Änderungen in einem größeren Software-Projekt ebenfalls in der Visualisierung erkannt werden können.
Außerdem kann der positive Effekt von farbigen Hervorhebungen durch einen methodisch sauberen vergleich der Systeme b) und c) verifiziert werden. Zuvor können in System c) außerdem die Verbesserungsvorschläge zur Hervorhebung von Änderungen an Packages integriert werden.

Anwendung zur Charakterisierung logischer Abhängigkeiten

Die durchgeführte Studie zeigt, dass in der Visualisierung Änderungen an kleinen Software-Projekten erkannt werden können. Die Studie untersucht nicht, ob die Visualisierung dabei helfen kann, die erkannten Änderungen miteinander in Beziehung zu setzen und dadurch logische Abhängigkeiten zu identifizieren und zu klassifizieren. Somit wurde bisher nur der erste Schritt dieses Prozesses untersucht.
Eine weiterführende Arbeit, die ihren Schwerpunkt auf die Identifizierung und Klassifizierung logischer Abhängigkeiten legt, sollte vor allem ein Datensatz verwenden, der Änderungen aus einer realen Entwicklungsarbeit widerspiegelt. Auf diese Weise ist sichergestellt, dass der gesamte Prozess anhand realistischer Daten erprobt und bewertet wird.

7 Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde ein Konzept zur Visualisierung der Historie OSGi-baiserteter Software-Projekten durch eine Inselmetapher entwickelt. Dieses wurde prototypisch in die virtuelle Umgebung der Anwendung IslandViz integriert und durch eine Nutzerstudie evaluiert.

7.1 Zusammenfassung des erstellten Konzepts

Der Schwerpunkt des vorgestellten Konzepts liegt auf dem Erhalt der mentalen Karte des Nutzers bei der Betrachtung der Software-Historie. Im Kontext der Arbeit beschreibt dieser Ausdruck, dass die Orientierung des Nutzers innerhalb der Darstellung erhalten bleiben soll, während sich die Bestandteile der Visualisierung aufgrund der Änderungen in der Software-Historie anpassen und verändern müssen. Dieses Ziel wurde auf den zur Verfügung stehenden Detailebenen unterschiedlich erreicht: Im Gesamtüberblick des Systems werden die Inseln des Archipels durch einen Algorithmus für dynamische Graphen positioniert. Dieser hält die Inseln in der Nähe ihrer vorherigen Positionen. Innerhalb der Darstellung eines Bundles als Insel ermöglicht ein adaptives Layout, dass jeder Region stets weitere Gebäude hinzugefügt werden können. Dies wird durch gesicherte Küstenzugänge und Wachstumskorridore der Regionen erreicht.
Die prototypische Implementierung setzt diese Konzepte um und ermöglicht eine lineare Navigation entlang der Historie eines Software-Projekts.

7.2 Ergebnisse der Nutzerstudie

In einer Nutzerstudie wurde der erstellte Prototyp mit einer Anwendung verglichen, die Software-Architektur ebenfalls als Inselwelt darstellt, dabei die Zeitpunkte aus der Historie jedoch nur unabhängig voneinander behandelt.

Die subjektive Bewertung der Nutzer zu ihrer Zufriedenheit ist bei beiden getesteten Visualisierungen ähnlich hoch. Die quantitative Bewertung zeigt jedoch, dass das für Software-Historie optimierte Konzept der Darstellung deutliche Vorteile bietet: Nutzer erkannten Änderungen innerhalb einer Insel im Prototyp durchschnittlich bis zu 50% schneller. Elemente, die auf einer der beiden Detailebenen geändert wurden, konnten 33% häufiger richtig benannt werden. Die dabei durch den Nutzer wahrgenommene gesamte Belastung wurde mit dem NASA-TLX bestimmt und war beim Bearbeiten der Aufgaben im Prototyp um 15% geringer.
Aus der Studie kann weiterhin gefolgert werden, dass eine zusätzliche farbliche Hervorhebung der Änderungen die Zufriedenheit mit der Visualisierung und die Effizienz bei der Bearbeitung der Fragen steigert.

7.3 Weitere Arbeiten

In zukünftigen Arbeiten kann einerseits die Implementierung der Visualisierung vervollständigt und verbessert werden. Dies bezieht sich sowohl auf die Möglichkeiten zur Nutzerinteraktion als auch auf die technische Umsetzung. Andererseits kann auch das Konzept zur Hervorhebung von Änderungen entsprechend der Ergebnisse der Nutzerstudie erweitertet und umgesetzt werden.
Nachdem die durchgeführte Nutzerstudie gezeigt hat, dass das Konzept geeignet ist, Änderungen in kleinen Software-Projekten zu erkennen, gibt es auch auf diesem Feld weitere Forschungsfragen. So muss zunächst gezeigt werden, dass die Visualisierung auch für größere Projekte geeignet ist. Schließlich kann untersucht werden, ob innerhalb der Visualisierung logische Abhängigkeiten identifizierbar und klassifizierbar sind.

Diese Arbeit bietet ein grundlegendes Konzept zur Visualisierung von Software-Historie durch eine Inselmetapher. Dieses kann optimiert werden, um Planungsaufgaben in der Software-Entwicklung zu unterstützen. Außerdem bietet das Konzept eine Basis für weitere Forschungen im Bereich der Software-Visualisierung.

Literatur

Balzer, M., Noack, A., Deussen, O. & Lewerentz, C. (2004). Software Landscapes: Visualizing the Structure of Large Software Systems. (S. 261–266).

Bangor, A., Kortum, P. & Miller, J. (2009). Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale. *J. Usability Stud. 4*, 114–123.

Beyer, D. & Hassan, A. E. (2006). Animated Visualization of Software History using Evolution Storyboards. In *2006 13th Working Conference on Reverse Engineering* (S. 199–210).

Brooke, J. (1995). SUS: A quick and dirty usability scale. *Usability Eval. Ind. 189*.

Caplan, S. (1990). Using focus group methodology for ergonomic design. *Ergonomics*, *33*(5), 527–533.

Cataldo, M., Mockus, A., Roberts, J. A. & Herbsleb, J. D. (2009). Software Dependencies, Work Dependencies, and Their Impact on Failures. *IEEE Transactions on Software Engineering*, *35*, 864–878.

Chapanond, A., Krishnamoorthy, M. S., Prabhu, G. M. & J., P. (2010). *Evolving Graph Representation and Visualization*. Department of Computer Science, Rensselaer Polytechnic Institute.

Collberg, C., Kobourov, S., Nagra, J., Pitts, J. & Wampler, K. (2003). A System for Graph-based Visualization of the Evolution of Software. In *Proceedings of the 2003 ACM Symposium on Software Visualization* (77–ff). SoftVis '03.

Diehl, S. (2007). *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Berlin, Heidelberg: Springer-Verlag.

Diehl, S. & Görg, C. (2002). Graphs, They Are Changing. In M. T. Goodrich & S. G. Kobourov (Hrsg.), *Graph Drawing* (S. 23–31). Berlin, Heidelberg: Springer Berlin Heidelberg.

Diel, S., Görg, C. & Kerren, A. (2001). Preserving the Mental Map using Foresighted Layout. In D. S. Ebert, J. M. Favre & R. Peikert (Hrsg.), *Eurographics / IEEE VGTC Symposium on Visualization*.

DLR-SC. (2020). IslandViz: A Tool for Visualizing Modular Software Systems in Virtual Reality. Zugriff 14. Februar 2020 unter https://github.com/DLR-SC/island-viz/tree/revised

Dresing, T. & Pehl, T. (2012). *Praxisbuch Interview, Transkription & Analyse Anleitungen und Regelsysteme für qualitativ Forschende.* dr dresing & pehl GmbH.

Eades, P. (1984). A heuristic for graph drawing. (Bd. 42, S. 149–160).

Eades, P., Lai, W., Misue, K. & Sugiyama, K. (1991). *Preserving the Mental Map of a Diagram* (Techn. Ber. Nr. IIAS-RR-91-16E). International Insitute for Advanced Study of Social Information Science.

Fruchterman, T. M. J. & Reingold, E. M. (1991). Graph Drawing by Force-directed Placement. *Softw., Pract. Exper. 21*, 1129–1164.

Gall, H., Hajek, K. & Jazayeri, M. (1998). Detection of logical coupling based on product release history. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)* (S. 190–198).

Gall, H., Jazayeri, M. & Riva, C. (1999). Visualizing software release histories: the use of color and third dimension. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)* (S. 99–108).

Hassan, A. E., Jingwei Wu & Holt, R. C. (2005). Visualizing historical data using spectrographs. In *11th IEEE International Software Metrics Symposium (METRICS'05)* (10 pp.–31).

Hofmeister, C., Nord, R. L. & Soni, D. (1999). Describing Software Architecture with UML. In P. Donohoe (Hrsg.), *Software Architecture: TC2 First Working IFIP Conference on Software Architecture (WICSA1) 22–24 February 1999, San Antonio, Texas, USA* (S. 145–159).

Hori, A., Kawakami, M. & Ichii, M. (2019). CodeHouse: VR Code Visualization Tool. In *2019 Working Conference on Software Visualization (VISSOFT)* (S. 83–87).

ISO/IEC-14764. (2006). *Software Enineering - Software Life Cycle Processes - Maintenance.* ISO.

ISO/IEC/IEEE-42010. (2011). *Systems and software engineering - Architecture description.* ISO.

Kennedy, R. S., Lane, N. E., Berbaum, K. S. & Lilienthal, M. G. (1993). Simulator Sickness Questionnaire: An Enhanced Method for Quantifying Simulator Sickness. *The International Journal of Aviation Psychology, 3*(3), 203–220.

Khaloo, P., Maghoumi, M., Taranta, E., Bettner, D. & Laviola, J. (2017). Code Park: A New 3D Code Visualization Tool. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)* (S. 43–53).

Lanza, M. (2001). The Evolution Matrix: Recovering Software Evolution Using Software Visualization Techniques. In *Proceedings of the 4th International Workshop on Principles of Software Evolution* (S. 37–42). IWPSE '01.

Lehman, M. M. (1996). Laws of software evolution revisited. In C. Montangero (Hrsg.), *Software Process Technology* (S. 108–124). Berlin, Heidelberg: Springer Berlin Heidelberg.

Lewis, J. (1991). Psychometric evaluation of an after-scenario questionnaire for computer usability studies: The ASQ. *SIGCHI Bull. 23*, 78–81.

Marcus, A., Feng, L. & Maletic, J. I. (2003). 3D Representations for Software Visualization. In *Proceedings of the 2003 ACM Symposium on Software Visualization* (27–ff). SoftVis '03.

McAffer, J., VanderLei, P. & Archer, S. (2010). *OSGi and Equinox: Creating Highly Modular Java Systems* (1st). Addison-Wesley Professional.

Menzies, T. & Zimmermann, T. (2013). Software Analytics: So What? *Software, IEEE*, *30*, 31–37.

Merino, L., Fuchs, J., Blumenschein, M., Anslow, C., Ghafari, M., Nierstrasz, O., ... Keim, D. A. (2017). On the Impact of the Medium in the Effectiveness of 3D Software Visualizations. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)* (S. 11–21).

Misiak, M. (2017). *Immersive Exploration of OSGi Based Software Architectures in Virtual Reality* (Master's Thesis, Technische Hochschule Köln).

Mockus, A. & Weiss, D. M. (2000). Predicting risk of software changes. *Bell Labs Technical Journal*, *5*(2), 169–180.

NASA, H. P. R. G. (1986). *NASA TASK LOAD INDEX (TLX) v. 1.0.* NASA Ames Research Center Moffett Field, California.

Oliva, G., Santana, F., Gerosa, M. A. & Souza, C. (2011). Towards a Classification of Logical Dependencies Origins: A Case Study. (S. 31–40).

Panas, T., Berrigan, R. & Grundy, J. (2003). A 3D metaphor for software production visualization. In *Proceedings on Seventh International Conference on Information Visualization, 2003. IV 2003.* (S. 314–319).

Powel, R. A. & Singel, H. M. (1996). Focus Groups. *International Journal for Quality in Health Care*, *8*(5), 499–504. eprint: http://oup.prod.sis.lan/intqhc/article-pdf/8/5/499/5215301/8-5-499.pdf

Schreiber, A., Nafeie, L., Baranowski, A., Seipel, P. & Misiak, M. (2019). Visualization of Software Architectures in Virtual Reality and Augmented Reality. In *2019 IEEE Aerospace Conference* (S. 1–12).

Steinbrückner, F. & Lewerentz, C. (2010). Representing Development History in Software Cities. In *Proceedings of the 5th International Symposium on Software Visualization* (S. 193–202). SOFTVIS '10. Salt Lake City, Utah, USA: ACM.

Steinbrückner, F. & Lewerentz, C. (2013). Understanding software evolution with software cities. *Information Visualization*, *12*(2), 200–216.

Tavares, A. L. & Valente, M. T. (2008). A Gentle Introduction to OSGi. *SIGSOFT Softw. Eng. Notes*, *33*(5), 8:1–8:5.

von Kurnatowski, L. (2018). *Visualisierung der Evolution von Softwarearchitektur* (Master's Thesis, Friedrich-Alexander-Universität Erlangen-Nürnberg).

Wettel, R. & Lanza, M. (2007). Visualizing Software Systems as Cities. (S. 92–99).

Yang, M. & Biuk-Aghai, R. P. (2015). Enhanced Hexagon-Tiling Algorithm for Map-Like Information Visualisation. In *Proceedings of the 8th International Symposium on Visual Information Communication and Interaction* (S. 137–142). VINCI '15.

Zimmermann, T., Weisgerber, P., Diehl, S. & Zeller, A. (2004). Mining Version Histories to Guide Software Changes. In *Proceedings of the 26th International Conference on Software Engineering* (S. 563–572). ICSE '04. Washington, DC, USA: IEEE Computer Society.

Anhang A. Fokusgruppengespräch: Leitfaden

Begrüßung

1. Dank für die Teilnahme
2. Kurze Erläuterung des Hintergrunds des Gespächs
   - Masterarbeit “Visualisierung von Software-Historie in VR”
     → Erweiterung von IslandViz
   - Ziel: Anforderungen an Darstellung von Software-Historie
     (sowohl direkt umsetzbar als auch langfristig)
3. Informationen zum Fokusgruppengespäch
   - Es ist erwünscht, aber nicht notwendig, auf Gesagtes gegenseitig einzugehen, gemeinsame Erfahrungen einzubringen, sich gegenseitig auf Ideen zu bringen usw.
   - Unterschiedliche Meinungen sind möglich und erwünscht
   - Es ist nicht nötig einen Konsens zu erzielen
4. Ablauf
   - Kurze Vorstellung
   - Nutzung von Informationen aus der Software-Historie in der Praxis
   - Anforderungen an Visualisierung allgemein und IslandViz im speziellen
5. Organisatorisches
   - Hinweis auf Tonaufnahme und Ausfüllen des Hinweisbogens zum Datenschutz
   - Regeln zum Gespächsablauf (Fragen gelten als Rahmen, andere Kommentare und Gedanken sind jederzeit möglich, für dringende Wortmeldungen kurz melden, ansonsten freie, normale Gesprächskultur)
   - Dauer ca. 2h
   - ca. nach der Hälfte Pause möglich
6. Rückfragen?

Zeit	Hauptfrage	Detailfrage	Notizen
15 min	**Block 0 - Vorstellung**		
	Werdegang, Momentane Aufgaben, Erfahrung		Schwerpunkt auf Software -Entwicklung, -Architektur, -Konzeption, Projektplanung
45-60 min	**Block 1 - Nutzung von Software-Historie**		
	Habt ihr schon Informationen über die bzw. aus der Softare-Historie benötigt?	Welche Aufgabe? Welche Informationen? Wie wurden die Informationen verwendet? Wie habt ihr die Informationen erhalten?	
	Könnt ihr euch andere Aufgaben oder Situationen vorstellen, in denen Wissen über die Software-Historie nützlich wäre?	Welche Informationen sind in diesen Fällen nötig?	z.B. Bug-Fixes? Refaktoring? Überprüfen, ob Planung umgesetzt wurde? längere Abwesenheit? Einarbeitung?

	Wie können Informationen über Software-Historie gespeichert werden?	Welche Aspekte werden in welchem Medium abgelegt?	z.B. Dokumentation? Wiki? Erinnerung? Git-Historie?
	Wie findet ihr die Informationen wieder?	Wie findet ihr den relevanten Zeitpunkt in der Historie? Welcher Zeitraum muss für die Arbeit abgedeckt sein? Wie navigiert ihr durch die Historie?	Zeitpunkt vs. Zeitraum? Von bestimmtem Zeitpunkt aus vorwärts / rückwärts? Vergleich von Zeitpunkten? Releases vs. einzelne Commits
	Pause bei Bedarf		
25 min	**Block 2 - Software-Visualisierung Allgemein**		Material: Bilder bestehender Arbeiten zur Visualisierung von Software (vgl. Kapitel 2.2.2)
	Habt ihr in den beschriebenen Situationen Visualisierungen genutzt um mit der Historie zu arbeiten? Bzw. hättet ihr euch Visualisierungen gewünscht?	Welche Visualisierungen wurden / würden genutzt? Welche Aspekte wurden Visualisiert?	z.B. Git-Commit-Graph? alte UML-Diagramme?

	Könntet ihr euch für die anderen genannten Beispiele Visualisierung als Unterstützung vorstellen?	Was soll dabei visualisiert werden?	
25 min	**Block 3 - Software-Visualisierung in IslandViz**		Material: Bilder alter Standt IslandViz, Darstellung von Konzept (vgl. Kapitel 4.3)
	Welche Anforderungen stellt ihr an die Erweiterung von IslandViz?		
	Wie könntet ihr euch vorstellen, euch durch die Historie zu bewegen?	An welcher Stelle im Zeitstrahl sollte die Visualisierung Starten?	ggf. Bezug auf Antworten aus Block 1

Tabelle A.1. Fragenkatalog Fokusgruppengespäch

Anhang B. Fokusgruppengespräch: Transkription

[...] (Vorstellung ausgelassen)

I: Vielen Dank schon mal. Das heißt, wir haben zwei alte Hasen, drei relativ Junge und ein Mittelding in unserer Runde. (...)
Es soll ja jetzt vor allen Dingen um Software-Historie gehen, (eine andere Studentin) hatte euch ja neulich zur Einarbeitung befragt. Jetzt geht es um Historie. Und ich stellen aber trotzdem eine ähnliche Frage und zwar: Fallen euch in eurer Arbeit Fälle ein wo, ihr euch tatsächlich mal die Software-Historie angeschaut habt und zu welchen Zweck habt ihr euch die angeschaut?

B1: Ja ich habe mir das durchaus hier und da angeschaut, ist aber kein täglicher Use-Case, dass das passiert. Eher selten würde ich sagen. (...)
Die Gründe dafür liegen zum einen darin, dass man vielleicht im Code hat, wo man sich relativ sicher ist, dass da schon mal was funktioniert hat. Und auf einmal irgendwelche Sachen passieren, die man nicht erklären kann. Dass man halt im Pfad zurück geht und schaut, was hat sich denn da in letzter Zeit getan. Woran können so Seiteneffekte vielleicht entstehen, oder was weiß ich. Das ist so der eine Anwendungsfall. Eine andere Anwendungsfall ist, dass man vielleicht versucht darüber noch Entscheidungen nachzuvollziehen, die mal vor einiger Zeit passiert sind. Und vielleicht dann bestenfalls in irgendwelchen Commit-Messages mit dokumentiert.

B2: Dann würde ich gleich weitermachen: Also aus verschiedenen Use-Cases. Das was ich tagtäglich tue ist, halt einfach wirklich in die kürzliche Historie rein zu schauen. Einfach um zu schauen, was hat sich zuletzt getan. Was haben die anderen da so commitet. Entsprechend mich auf den neusten Stand bringen letztendlichen. Dann gibt es eben noch tatsächlich das, wo ich ein bisschen länger in der Historie vielleicht zurückschaue. Das ist dann zum Beispiel wenn ich in einen Code-Bereich reinschauen, wo ich lange nicht dran war oder noch gar nicht dran war, um mal eben ein bisschen zu gucken, wann ist da zuletzt was passiert. Erst mal bei der Produktivität. Und dann ggf. noch wer daran war. Insbesondere wenn man die Person natürlich kennt und einschätzen kann. Das gibt auch nochmal Rückschlüsse, was mit dem Code ist. Und tatsächlich so wie B1 auch schon erwähnt hat

die Commit-Messages spielen eine sehr große Rolle. Also weniger okay, welche Klassen wurden jetzt wie oft angepackt, das sagt mir eher wenig. Sondern halt wirklich okay was ist die Beschreibung. Das ist jetzt nicht genau die Frage, aber gute Commit-Messages sind extrem wichtig. Da lege ich sehr Wert drauf. Ja ich glaube, das wären die Use-Cases jetzt erstmal.

I: Also wie gesagt, wenn irgendwas zwischendurch einfällt immer gerne dazwischen werfen.

B1: Also das vielleicht nur ergänzend: Weil ich jetzt eben sagte, dass man versucht auch irgendwie Entscheidungen nachzuvollziehen oder irgendwie darüber vielleicht auch Sachen zu verstehen. Da spielen natürlich auch die Personen, die irgendwie am Code gearbeitet haben, immer eine große Rolle. Gerade wenn sie noch anwesend ist, dann kann man da hingehen und fragen zum Beispiel. Aber man weiß ja auch im Team, wer grob an welchen Ecken arbeitet oder gearbeitet hat. Selbst wenn Leute nicht mehr da sind, lässt das auch Rückschlüsse zu.

B3: Ich finde es auch noch unterstützend mit Entscheidungen nachzuvollziehen. So ganz konkret im Sinne von: Ich gucke mir irgendwie einen Code an und habe dann eine Stelle gefunden, bei der ich halt keine Ahnung habe, warum die jetzt da drin ist. Und dann hilft es auch teilweise sich die Historie anzugucken und dann zu schauen, wann ist das denn reingekommen. Und dann auch mit Hilfe der Commit-Messages das vielleicht nachvollziehen zu können warum ist es überhaupt reingekommen.

B1: Also Commit-Message ist bei uns ja noch so, dass sie immer noch zu einem Mantis-Issue verlinkt sind. Vielleicht ist das als Background auch noch wichtig zu wissen. Das heißt darüber kommt man auch noch zu weiterer Dokumentation. Das hilft dann auch unglaublich viel.

B4: Ich wollte eigentlich nur ergänzen, dass ich es auch wirklich zur Fehlersuche nutze. Entweder ist das vielleicht ein neues Feature was noch gar nicht lange drin ist und es ist deshalb einfach noch nicht fertig. Oder, oft sieht man ja dann auch an so einer Commit-Historie: ok da ist jetzt etwas umgestellt worden und dabei wurde vielleicht was vergessen oder so. Dadurch ist jetzt dieser Bug entstanden.

B2: Vielleicht ein Use-Case, der noch nicht genannt wurde, ist: Ich benutze auch die Historie wirklich einfach auch Subversion, zum Beispiel das Commit-Log von einem Branch zum Beispiel mache ich auch öfters, um zu schauen auf welchem Merge-Stand der ist. Also wann ist zuletzt welcher andere Branche, auch der Trunk und so, dann halt reingemergt worden. Oder umgekehrt halt wann, zu welchem Zeitpunkt ist der Branch abgespalten worden.

I: Wenn ihr diese Information euch anschaut, sucht ihr dann vor allem direkt in dem SVN, in dem Git, dass ihr dort auf der Seite seid, oder auf welcher Plattform findet ihr diese Information, die ihr sucht?

B2: Also ich gehe über Tortoise-Git, mit dem ich sowieso arbeite, und dann mache ich es auch meistens eben so, dass ich dann entweder auch über einen Ordner, den ich sowieso ausgecheckt hab, oder dann eben in dem Repo-Browser gezielt auf einen Unterbaum, einen Branch oder wenn es halt Sinn macht auf dem Gesamtprojekt oder sogar auf dem gesamten Repository. Also ich wähl' mir schon selektiv welchen Teilbaum quasi ich die Historie sehen möchte.

I: Und dann zum einen eben im Browser, dass man da so die Liste schön untereinander hat oder über die Konsole, dass das dort aufgelistet, oder (...)?

B2: Also über die Gui von dem Subversion, von dem Tortoise-SVN. Also Webbrowser benutze ich dafür praktisch überhaupt nicht. Und Konsole: Ich wüsste wie ich das machen würde, aber es gibt keinen Grund. An der Stelle ist es nicht schneller oder nützlicher als die GUI.

B1: Also ich mach das im Endeffekt genauso, darüber hinaus. Zum Teil wenn es sich um ganz spezielle, kleinere Code-Abschnitt handelt, auch aus Eclipse heraus, also direkt aus der Entwicklungsumgebung. Und auch unser Mantis hängt halt die Commit-Messages und Change-Logs an die Issues an. Und manchmal macht es auch Sinn, einfach da schnell rüber zu schauen. Geht manchmal noch schneller.

B2: Also der Weg über den Mantis, da wird umgekehrt quasi mehr geschaut, welche Commits sind sowieso schon auf diesen Issue gemappt, das nutze ich auch sehr viel. Und was ich auch teilweise sogar auch noch tue, ich hatte eben erwähnt, wenn ich Unterbäumen des Repositories durchsuche, dann schaue ich schaue mir dann auch durchaus öfters mal gezielt an, ich habe dann immer eine Datei wo ich dann wissen will was ist damit. Und dann mache ich dann wirklich Rechtsklick auf die Datei und schau mir dann wirklich das Subversion-Log für diese Datei an wo ich dann und wann ist diese Datei, in welchen Commits ist die angepackt worden.

B3: Wobei, das mache ich dann tatsächlich meistens schon in Eclipse selbst.

B2: Genau, das ist Geschmackssache.

B3: Weil man da gerade eh schon ist.

B4: Ja, ich mache es auch meistens über diese History-Ansicht in Eclipse. Und da kann man ja auch einfach zwei Versionen diffen und sich anschauen was hat sich da geändert.

Das ist meistens auch ganz praktisch.

I: Das wäre dann tatsächlich auf Code-Ebene, dass du dann das diffst.

B4: Ja, genau.

I: Und du dann siehts welche Zeile wurde geändert.

B4: Ja, also dass ist dann nicht nur unbedingt zwischen den neuesten Versionen, sondern du kannst halt auch sagen, ich will jetzt die aktuelle Version mit, ja, zehn Versionen vorher vergleichen oder so. Wenn ich halt den Verdacht habe, dass da irgendwo was reingekommen ist.

B1: Oder auch die neunte und zehnte Version vorher, das ist manchmal auch wichtig, dass man genau zu einem Commit das Diff bekommt.

B2: Vielleicht, fällt mir noch kurz zu Diff und Historie und so. Ich weiß nicht wie sehr das jetzt hier reinpasst in den Scoop. Aber ich mach's dann auch teilweise so, wenn ich z.B. einen Branch hab, den jemand anders gemacht hat, einen Feature-Branch, dann mache ich das teilweise auch so, dass ich dann den aktuellen Stand gegen den Zeitpunkt, wo der Branch abgespalten / eröffnet wurde diffe. Also das gesamte Diff, das überhaupt in diesem Branch gemacht wurde, mir anschaue.

B3: Und, was ich auch in Eclipse noch häufiger nutze ist, SVN-Blame-Phrase/Annotate, wie auch immer man das nennt. Einfach weil dann auch direkt zu jeder Zeit angezeigt wird, wer die Zeile zuletzt angepackt hat. Das spielt dann auch wieder mit darein, herauszufinden, wer daran gearbeitet hat. Ist teilweise natürlich, wenn man Merging-Commits hat, nicht so unglaublich praktisch mehr, weil dann hat halt derjenige der gemerged hat, halt

I: hat alle Zeilen genommen.

B3: Genau. Und da muss man sich halt manuell ein bisschen durcharbeiten, aber ab und zu hilft es halt einfach und es ist halt einfach zwei Klicks entfernt, aus dem normalen Arbeitsalltag.

B2: Und vielleicht noch ergänzend: Auch bei diesem, wenn ich das mache, mir den Branch anzuschauen, ich will ja wissen was hat irgendwer in diesem Branch gearbeitet, da hab ich dann dieses Problem, wenn jemand nochmal, z.B. den Trunk dann reingemerged hat, ist natürlich ein riesiges Diff was aber völlig irrelevant ist. Das wäre jetzt zur Frage Was könnte bei der Arbeit mit der Historie besser sein wäre das sicherlich ein Punkt, wenn ich sage okay so Merges möchte ich jetzt automatisch gar nicht sehen weil, die interessieren mich gerade nicht.

I: Du hattest vorhin gesagt, du willst den Commit 10 mit dem Commit 8 vergleichen, sozusagen. Woher weißt du denn, dass der Commit 10 der Relevante ist, wie findest du das heraus?

B4: Das weiß ich eben nicht. Sondern ich habe dann vielleicht so eine Idee auch anhand der Commit-Messages wo, zumindest so in welchem Zeitraum sich da was geändert hat und dafür nutze ich das halt gerne, dass ich dann erst einmal großflächiger gucke. Weiß nicht, zwischen Commit 2 und Commit 10 was hat sich da geändert und wenn ich da dann sehe da ist es drin dann kann ich ja genau herausfinden wo in der Zwischenzeit das passiert ist.

I: Also du klickst dich dann praktisch kleinschrittig, Schritt für Schritt, Commit für Commit durch, nachdem du den Bereich eingegrenzt hast? **B4:** Ja, wobei das schon ein spezieller Fall ist, also meistens ist es schon anhand der Commit-Messages relativ leicht dann so 'n paar rauszufinden die jetzt in Frage kommen.

B2: Was auch noch interessant ist, wir haben jetzt viel von den Commit-Messages gesprochen, Was ich auch nutze ist gerade auch bei Branches, aber auch teilweise mal 'n Trunk und so, ist, dass ich auch einen Bereich von Commits markieren und mir dann anschauen entsprechend welche Dateien auch in diesem Commit-Block quasi angepackt wurden. Oder auch einzelne Commits, dass ich auch die Historie durchgehe und dann, klar die Commit-Message lese ich natürlich auch, aber dann wirklich auch gucke ok in welchem Code-Bereich war das dann. Das ist aber sehr speziell wann ich das nutze, also das ist nicht so häufig, dass ich das brauche, und dass es nützlich ist.

I: Aber was wäre so ein Fall dass du wissen willst, welche Dateien wurden alles angefasst?

B2: Mhh, das ist schwer zu beschreiben. Da kann ich jetzt keinen nennen, das richtig beschreiben. Das ist sehr speziell.

I: Ok, aber manchmal ist es auch tatsächlich einfach nur relevant welche Dateien sind in dem Zeitraum geändert worden sind?

B2: Müsste ich jetzt drüber nachdenken, also kann ich jetzt so gar nicht sagen.(...) Also, es kann zwar z.B. mal nützlich sein, wenn ich weiß, jemand hat irgendwo gearbeitet und dann weiß ich ein bestimmter Code-Bereich ist völlig klar, dass er daran gearbeitet hat, das ist ganz neuer Code, der da entsteht. Und dann will ich aber zum Beispiel schauen, hat er dafür andere Bereiche im System angefasst? Musste er irgendwo anders noch Schnittstellen erweitern oder anpassen oder Utilities hinzufügen also so was dann eben. Also mehr so aus Umkehrprinzip.

B1: Also würde ich auch genauso zustimmen, einen Anwendungsfall gibt es auch schon mal, dass man halt sagt, man hat eigentlich ganz klar umrissen die Bereiche, wo gearbeitet wird aber manchmal rutschen bei Commits vielleicht auch mal hier und da Dateien mit rein und das kann man darüber halt sehr schnell sehen und ausschließen ob darin ein Fehler sein könnte.

B4: Mir fällt grad noch ein anderer Fall wo ich das auch öfter nutze. Und zwar wenn ich einen größeren Branch mergen will in den Trunk oder umgekehrt. Und dann tauchen da ja schon mal Konflikte auf. Und ich sehe ja im Branch hat sich das in die Richtung geändert und im Trunk wurde irgendwas anderes gemacht und dann muss ich eben auch versuchen nachzuvollziehen warum wurde das so geändert. So unterschiedlich. Und was ist denn jetzt so zu sagen die richtige Version oder die, die ich jetzt in meinem Branch haben will. Da such ich auch teilweise dann länger in der Historie rum, um herauszufinden wo so eine Änderung herkommen und was da eigentlich passiert ist.

I: Das sind in dem Fall relativ detaillierte Änderungen wahrscheinlich auch, dass dann kleine Zeilen oder einzelne Funktionalitäten anders sind?

B4: Ja, manchmal das. Aber teilweise kommt es auch vor, dass zum Beispiel irgendwo Code aufgeräumt wurde und dann heißen Klassen plötzlich anderes im Trunk oder gibt's gar nicht mehr und dann muss ich erst mal rausfinden: ok, die habe ich in meinem Branch aber noch benutzt. Was ist denn damit passiert und wie muss ich jetzt meinen Code umstellen, damit das wieder passt.

B1: Also ich glaub, Umfang - Art und Umfang, also vom Detailgrad her, Von Änderungen die man sich da anschaut, das kann alles sein. Das kann wirklich von einzeilig bis halt wirklich, keine Ahnung, kompletten Bundles, oder, weiß nicht, also das kommt total darauf an was man sich anschaut.

B2: Genau, und eine Sache, die eben auch schon ein bisschen durchkam war auch der Punkt. Änderungen auch checken wo eben so Sachen außerhalb vom Haubtarbeitsbereichen liegen. Dann geht es auch teilweise wirklich darum, in der Historie wirklich auch bisschen einschätzen zu können. War die Änderung jetzt absichtlich oder war das ein Commit-Unfall, weil das passiert in der Praxis gar nicht so selten. Also der Fall den ich dann nicht suchen will, weil das fällt meistens früher auf wenn beim Commit Dateien vergessen wurden. Das fällt meist auf, weil dann irgendwas nicht funktioniert. Aber was halt schon häufiger passiert und nicht bemerkt ist, dass dann eben ausversehen zu viel commitet wurde. Entweder sind Dateien ins Repository gerutscht, die da nicht reingehört einfach oder es sind aus Versehen, was weiß ich, jemand hat seine lokale Config geändert

zum Testen und hat dann ausversehen das Ding mit commited. Und gehts natürlich, dann hat man plötzlich eine andere Config im Projekt und denkt sich dann huch, warum das jetzt und dann gucke ich in die Historie okay war das ein absichtlicher Commit der Datei, oder war das bei einem völlig irrelevanten Commit mit reingerutscht. Das kommt auch mittelhäufig vor.

I: Wie würde das denn genau aussehen, wie du danach suchst?

B2: Dann geh' ich auf die Config-Datei zum Beispiel, guck mir da die History an und guck dann die Commits schonmal, kann ich an der Commit-Message schonmal relativ gut erkennen war da irgendwas hier üpdated Configurationöder irgendwas, was ganz anderes.

I: Ok. Wenn ihr so in der Historie zurückgeht um die verschiedenen Branches, Änderungen im Branch zu sehen oder um zu sehen wer hat was wann angefasst, Wie weit sind da so die Zeiträume, die ihr das zurück geht? Kann man das schätzen, oder kann man da was zu sagen?

B4: Ich glaube das ist extrem unterschiedlich. Weil es gibt Sachen, die wurden seit Jahren nicht angepackt. Und dann guckt man sich da vielleicht in den letzten Commit an. Und teilweise sind es auch ganz aktuelle Sachen.

B2: Also auf einen Zeitraum wüsste ich auch nicht, wie man's einschränken sollte. Aber es sind schon meistens die relativ letzten Änderungen. Also es ist nicht so oft, dass ich jetzt wirklich in die Urgeschichte des Projektes zurückgehe, das ist meistens völlig irrelevant.

I: Weil du hattest z.B. vorhin auch gesagt, um dich auf dem neuesten Stand zu bringen. Was ist da dann so... ne Woche, 2 Wochen, einen Monat?

B2: Also meistens arbeiten wir entweder halt im Trunk oder halt irgendwie in Branches und die sind auch meisten noch relativ aktiv und dann geht es dann meistens wirklich so seit dem letzten Arbeitstag halt meistens. Also selten mehr als eins zwei Wochen.

B1: vielleicht aber da, weil ja eben auch die Art und Weise wie man auf die Historie zugreift entscheidend ist, ne Zeit lang, im Moment kommt das irgendwie nicht, haben wir ja auch E-Mails bekommen so über Änderungen zum Teil. Zum Beispiel an Mantis-Issues. Das ist ja auch eine Art von Historie jetzt nicht vom Source-Code aber vom Entwicklungszyklus. Und das ist auch eine Möglichkeit sozusagen ein bisschen im Blick zu behalten was passiert gerade. Ich glaube es gibt doch einfach die Informationen zum Teil einfach redundant auf verschiedenen Wegen. Und das nutzt irgendwie auch jeder für sich.

I: Das heißt, du meinst, dass man nicht nur in der Historie des Programms sozusagen guckt sondern auch wann sind welche neuen Issues dazugekommen?

B1: Oder wann wurden die Issues bearbeitet und wie gesagt an die Issues werden halt auch die Commit-Messages und die Change-Sets vom Source-Code angehangen. Und man kann halt auch einfach auf einen Mantis-Issue klicken und gucken was hat sich denn im Code sozusagen im Bezug auf diesen Issue getan. Und wie gesagt das kann man in Mantis über die Weboberfläche sehen, je nachdem wie man das konfiguriert hat, kann man sich auch per E-Mail darüber informieren lassen. Man kann halt über den Repo-Browser gehen. Also da gibt es auch die unterschiedlichsten Möglichkeiten und damit auch Arbeitsweisen, glaube ich.

B2: Wobei das über den Issue zu gehen natürlich keine generische Sache ist, die immer funktioniert, weil natürlich da muss ich wissen, welchen Issue ich suche. Und genau dieses mich auf den Stand zu bringen, ist das wirklich kein vorgegebenes, das funktioniert. Aber, wie B1 schon sagte, jedem Fall.

B3: Um sich auf den aktuellen Stand zu bringen, würde es natürlich auch helfen, einfach durch die letzten Issues zu gehen. Ich mein, mache ich teilweise auch so, einfach mal, wenn ich so eine Woche nicht reingeschaut hab oder drei Wochen im Urlaub war, einfach mal die Issues angucken. Die werden ja sortiert nach den letzten Aktivitäten. Und dann einfach mal so durchgucken was da in letzter Zeit so passiert ist.

B2: Also, wenn wir jetzt hier wirklich den Issue-Tracker auch mit in den Scope hier, was Historie angeht, mit reinnehmen. Das nutze ich auch sehr viel, häufiger als die Subversion-Historie. Ich schaue wirklich ständig auf die Issues und genau wie Alex gerade sagte, dann eben die neuesten bearbeiteten Sachen, auf die Commits kamen, poppen nach oben, und dann sehe ich automatisch, an welchen hat sich was getan.

B1: Wobei ich glaube, ich nutze das auch mehr als die Historie aus dem Repo. Aber ich glaube, von einer anderen Sichtweise. Also das ist mehr so diese planerische Sichtweisen und im SVN nutze ich mehr so diese Code-Sichtweise in Richtung debugging. Und halt anderen Scope. Also wenn es natürlich weg von konkreten Issues geht und mehr für, also sag ich jetzt mal, Entscheidungs-Nachvollziehbarkeit oder auch Bugs, die vielleicht jetzt nicht einem Issue zuzuordnen sind, sondern man stellt irgendwas irgendwie in seiner Arbeit fest, da funktioniert was nicht oder da hat sich was getan. Und weiß gar nicht warum, kann man natürlich nicht über nen Issue gehen. Gibt viele Möglichkeiten.

I: Weil wir jetzt so viel von Issues gesprochen haben, würde mich auch interessieren wie findet man den Issue, den man sucht? Also habt ihr die Nummern im Kopf, oder gibts da ein Stichwort Verzeichnis wie findet man den Issue, von dem aus man dann ja weitergeht?

B2: Kommt ganz darauf an wo man herkommt. Also wenn ich jetzt weiß okey in dem

thematischen Bereich trats auf, dann mache ich wirklich in dem Issue-Tracker einfach nur ne Suche. Einfach mit nem Schlüsselworte und dann ja meist so, weiß nicht, mit einem typischen Schlüsselworte so zehn bis zwanzig Treffer und die guck ich kurz durch und dann finde ich den richtig schon. Es macht auch glaub' nicht viel Sinn, wenn das viel spezieller wäre, weil Issues sind einfach oft auch entweder ein bisschen unscharf oder ein bisschen redundant. Also ganz drum herumkommen, da so eine Liste durchzugucken würde man gar nicht vermeiden können, glaube ich.

B2: (unv. Sprecher spricht sehr undeutlich) denn, wenn es aktive Issues sind, die sehe ich ja auch genau, weil sie oben in der Liste sind. Oder teilweise gucke ich auch in der Commit-Historie, das ich sehe, daran wird gerade gearbeitet und dann gucke ich mir die Nummer an.

B1: Aus planerischer Sicht ist es ja auch noch so, dass wir gerade wenn's so in Richtung eines Releases geht und über die Issues versuchen wir ja auch sozusagen die Roadmap abzubilden. Und dann hat man Release-Targets und wenn das halt konkret wird, dann kann man ja auch danach gut filtern und dann sieht man sozusagen genau jetzt in dem Scope für die nächsten paar Wochen, was muss ich da noch tun. Und da kann man darüber versuchen hier mal diesen planerischen Überblick zu bekommen und dann gehts gegebenenfalls dann irgendwann von Mantis zu SVN, um da weiter zu gucken.

I: Das heißt, die Issues sind auch verschiedene Releases oder Arbeitspakete zugeordnet.

B3: Und es ist auch so, um den richtigen Issue zu finden, ich würde sagen, in 95 % der Fälle sucht man nach Issues, die in den letzten, keine Ahnung, sechs Monaten maximal angepackt wurden. Die restlichen fünf Prozent verteilen sich dann auf die Issues, die zwischen vor einem halben Jahr und vor 15 Jahren behandelt wurden. Und deswegen, wenn man einfach in dieser nach Zeit sortierten Liste oben anfängt ist man meistens relativ schnell bei dem, den man sucht. Und dann wie gesagt 20, 30 per Hand durchgehen, das ist ja auch schnell gemacht.

I: Wie sieht es aus: benutzt ihr die Software Historie auch manchmal, dass ihr gerad' wenn man jetzt vielleicht 'ne neuere Komponente entwickelt oder eine neue Funktionalität dazu, dass man sich vorher vornimmt: Ich will das entwickeln und dann hinterher noch mal gucken, in welchen Schritten hab ich das gemacht. Das man sich so noch mal, dass sozusagen für sich bewertet oder schaut habe ich das tatsächlich so umgesetzt wie ich das wollte oder bin ich da von meinen Plänen abgerückt oder so. Kommt das vor so im Entwicklungsprozess?

B2: Ist mir noch nie passiert, weil gerade, wenn ich da gerade dran gearbeitet habe, dann

weiß ich das. Und, wenn es länger zurückliegt, also den Fall gab es bisher noch nicht.

B3: Was ganz interessant ist, wenn es so Teilprojekte gibt, die halt auch auf Branches entwickelt werden und man sich da nicht sicher ist auf welchem Stand ist das Teilprojekt jetzt. Da, gerade wenn das von Kollegen gemacht wurde, die jetzt nicht mehr da sind und es da teilweise ganz praktisch in diesen Branch zu gehen und da dann danach zu schauen wie weit ist das eigentlich, auf welchem Stand ist das. Ist das man müsste es nur noch einen Trunk mergen oder ist da noch viel zu tun. Das ist ein bisschen wie die Projektplanung zu rekonstruieren.

B4: Ich habe es schon mal genutzt, so beim Berichtschreiben weil ich nicht mehr genau wusste, wann wurde denn dieses Feature entwickelt und dann hab ich dann tatsächlich mal in der Historie geguckt so wann hab ich das denn commitet. Aber das ist eigentlich auch ein Sonderfall, ist jetzt nichts, was ich regelmäßig mache.

B2: Wir haben auch ganz vereinzelt mal so Fälle. Also in der Regel haben wir auch Change-Logs einfach. Die wir sowieso pflegen und anlegen für jedes Release und da findet man sowas meistens schneller und effizienter. Aber wenn dann so Sachen sind, gerade so technische Dinge, die mehr intern passiert sind, die halt fürs Change-Log nicht relevant sind für den Benutzer. Und dann wissen wir, wann ist das geändert worden, wann ist das reingekommen, dann gehe ich da manchmal schon auf die Suche. Aber das kommt extrem selten vor. Ich würde so sagen, im Schnitt vielleicht einmal im Jahre oder so.

B3: was ich tatsächlich ab und zu mal gemacht habe ist, wenn das so ne Woche oder zwei waren, wo ich relativ viel Kleinkram gemacht habe. Und es dann am Anfang nächster Woche in die Monatsrunde geht und ich dann nochmal durchgehe möchte, ok ich hab jetzt die zwei Wochen jede Menge Kleinkrams gemacht, ist irgendwas davon jetzt relevant, was man vielleicht mal kurz sagen sollte, was ist jetzt anders als vorher. Dann gehe ich tatsächlich mal durch meinen Commits und gucke mal, was ich jetzt eigentlich getan hab. Das ist natürlich bei größeren Aufgaben nicht wirklich relevant, weil da erinnert man sich ziemlich genau dran, was man getan hat. Aber wenn es jetzt halt eine Menge kleinerer Bugfixes sind, dann kann man da drüber schauen.

I: Du hattest vorhin gesagt im Code aufgeräumt hat jemand. Wie geht ihr da so ran, wenn Ihr sagt ihr wollt jetzt mal ein Refactoring machen, im Code aufräumen. Spielt da die Software Historie eine Rolle, dass ihr euch erst mal so anschaut, was ist in den letzten Wochen oder in den letzten Monaten eigentlich alles an Müll aufgekommen oder nimmt man sich einfach so den Code-Stand und arbeitet drauf los? Wie funktioniert das?

B2: Ich persönlich, ganz klar Letzteres. Also ich mach ziemlich viel mit Refactoring. Also

zum einen generell so die Gesamtarchitektur von so einem System, also jetzt gerade RCE speziell, hab ich stark im Blick und auch so, also ich refactore relativ häufig. Ich kann mich jetzt ad hoc an keine Situation erinnern wo ich da in die Historie geschaut hätte. Ich schaue mir immer den Jetztzustand an.

B3: Wobei da spielt bei mir halt auch dann relativ häufig rein, dass wenn ich refactore, was ich am Anfang gesagt hatte, dass ich dann mehr da durchgehe jetzt nicht weiß ob irgend so ein Code-Teil tatsächlich noch gebraucht wird. Es kann ja auch sein, dass der noch ein Artefakt ist, aus irgendeiner älteren Zeit ist und früher mal gebraucht wurde. Und jetzt ist aber der Teil weggefallen, für den man ihn eigentlich gebraucht hätte. Da habe ich dann vor dem Refactoring, also das ist dann ja kein Refactoring mehr, das ist dann mehr eine Änderung, da würde ich dann auch nochmal hingehen und nochmal nachgucken, warum es ursprünglich reingekommen und ist der Teil jetzt tatsächlich weg, wofür man den ursprünglich hat.

B2: Ja, wobei, aus meiner Sicht würde ich da eher sagen, da guck ich meinst nicht in die Historie, weil mir die meistens nicht besonders viel bringt. Also ich guck dann eher wirklich eher dann den aktuellen Code an und hoffe dann insbesondere, dass da dann auch Kommentare dranstehen, also z.B. wenn jemand jetzt ein Code-Teil, der einen anderen Code-Teil benutzt, löscht und das war der letzte Nutzer von diesem anderen Code-Teil, dann würde ich hoffen, dass da dann einen Kommentar dran stehen würde, "Hier haben wir noch drin gelassen, weil könnten wir noch brauchen"Wenn das nicht dran steht, dann bringt mir auch die Historie nicht viel, weil dann kann ich ja nicht in den Kopf von dem schon, der damals das gelöscht hat, ob er das bewusst dringen lassen hat oder er hat's einfach nur übersehen oder vergessen oder keine Zeit gehabt. Das heißt, die Historie sag mir nichts.

I: Wir haben jetzt relativ viel Historie auf Code-Ebene besprochen. Mich würde interessieren, das ist vielleicht sogar gerade, wenn du sagst auf System Ebene, ob ihr auch an der Architektur (weil jetzt geht es ganz langsam Richtung IslandViz, weil das ist ja mehr so auf Architektur Ebene) ob ihr da auch Software Historie, also die die Historie der Architektur sozusagen, ob die relevant ist und ob die für Aufgaben relevant sein könnte.

B2: Da müssten wir jetzt erst mal den Begriff Architektur erst mal klären. Weil das wird nämlich gerade, kann ich auch gerad mal zu Protokoll geben, gerade im Umkreis der IslandViz aus meiner Sicht sehr, sagen wir mal, großzügig verwendet dieser Begriff. Das wird teilweise auch wirklich dafür verwendet, wenn es wirklich nur heißt, so wir haben jetzt hier Packages und da sind Klassen drin, da wird dann gesagt, so wir haben jetzt hier

die Architektur visualisiert. Das ist für mich von Architektur noch sehr, sehr weit entfernt. Das würde ich persönlich sagen, das ist die Software Struktur. Das ist von Architektur noch sehr weit entfernt.

I: Wie definiert sich denn in deiner Arbeit Architektur?

B2: Wenn man das definieren könnte, wäre die Welt schön. Aber es ist wirklich sehr viel komplexer. Also sehr viel mehr Interaktion, Dynamik, sehr viel mehr Details, also einfach nur die Klassen- und Packet- Struktur, das sind die absoluten Basics.

I: Was würde denn zum Beispiel dazu kommen? Weil das, was die IslandViz als Struktur hat, ist ja mehr oder weniger nur Abhängigkeiten auf Package-Ebene. Was würde für dich zum Beispiel, du musst jetzt nicht alles aufzählen, noch dazukommen?

B2: Jetzt habe ich ein großes Thema angesprochen.

I: Vielleicht kann auch jemand dir noch helfen. Also geht's da um Aufruf-Hierarchien von Funktionen untereinander oder geht's da um welche Klasse hat welche andere als Member oder geht es um ganz was anderes.

B2: Also, wenn ich an Architektur jetzt erst mal denke, wenn ich jetzt mal nicht an IslandViz denke, so wie sie jetzt ist, sondern einfach von Architektur rede, dann geht es um Strukturen, dann geht es um Patterns, dann geht auch sehr viel einfach um Interaktion. Also wenn man allein schon sowas simples denkt wie 'n UML Sequenz-Diagramm z.B. Allein sowas ist ja in der jetzigen IslandViz überhaupt nicht vorhanden zur Information. Also einfach wie arbeitet das System. Das ist die entscheidende Information und nicht wie ist das, was die Software tut, jetzt auf Code-Dateien runtergebrochen.

I: Also Interaktion im Sinne von wie interagieren die einzelnen Teile des Programms und nicht die Nutzerinteraktion.

B2: Genau, also mehr die Funktion, nicht die Code-Struktur.

B1: Ich würd' vielleicht sogar so weit gehen, bei der IslandViz sagt man ja immer hat die Möglichkeit OSGi Software zu visualisieren. Ich finde, dass OSGi schon ein Teil der Architektur ist.

I: Kannst du das nochmal irgendwie verdeutlichen?

B1: Ja, also OSGi ist ja schon eine Ebene sozusagen, die definiert wie Code-Teile letztendlich miteinander kommunizieren können. Das heißt, das ist ja sozusagen eine Schicht, in dem Fall würde ich sagen darunter. Und das wird jetzt hier sozusagen bei der IslandViz, wo man sozusagen von Architektur-Visualisierung spricht schon irgendwie vorausgesetzt. Und deshalb, also da bin ich eher bei B2, das ich sage, okay das ist halt die Struktur letzt-

endlich, weil man hat lediglich Abhängigkeiten, die man visualisiert. Aber OSGi macht ja noch viel mehr letztendlich. Und das fehlt, diese Information fehlt halt einfach komplett in der Visualisierung. Und ich sage mal so, dass, wenn man jetzt RCE z.B. betrachtet, was ja immer gerne als Beispiel genommen wird, dann finde ich, wenn da jemand neu ins Team kommt dann liegt die Komplexität des Ganzen nicht nur darin, dass wir ganz viele Bundles haben die irgendwelche Abhängigkeiten haben, sondern dass das Ganze halt auf OSGi basiert und das hat irgendwie seinen speziellen Eigenschaften. Und das muss man irgendwie auch verstehen und kennen. Also was ist OSGi. Ich glaube damit fängt es bei vielen Entwicklern schon an, erstmal zu verstehen was das ist. Und dann sind wir erst bei dem Schritt wo man vielleicht über Abhängigkeit nachdenkt. Das ist so mein Eindruck zumindest.

B3: Und was z.B. auch jetzt in der IslandViz auch nicht dargestellt wird ist, also soweit ich weiß, ist so die Stärke der Abhängigkeiten, klar man kann sehen mein eines Bundle benutzt irgendwas, irgendwo abhängige Sachen aus dem anderen Bundle, aber das ist ja ein qualitativer riesiger Unterschied. Ob das jetzt so 'n Logger ist, wo ich halt einfach nur so 'n paar Logging-Botschaften reinwerfe, und ansonsten ich könnt den auch einfach rausnehmen und das tut sich nicht. Oder die beiden Bundles kommunizieren ganz stark miteinander und sind voneinander so abhängig, dass das eine seine Aufgabe nicht ohne das andere wirklich ausführen können. Das ist was, was in der IslandViz gar nicht ist, ich glaub, das ist was, worauf du (B2) eben auch hinauswolltest.

B2: Das ist ein guter Punkt, den du ansprichst, weil gerade auch dieses ist ja auch ein ganz häufiges Problem in der generellen Software Visualisierung. Da wird ja ganz oft gemacht. Z.B. man hat mal gesagt, ok, wir nehmen die Anzahl der Dependencies, die Anzahl der Code-Referenzen, und wir nehmen jetzt einfach, wir definieren jetzt einfach mal die Anzahl der Referenzen als Stärke der Beziehung. Aber aus architektonischer Sicht, aber ist das völlig irreführend. Das hat damit unter Umständen überhaupt nichts zu tun. Genau wie du sagst: Logging. Da haben die alle scheinbar architektonisch total viel mit dem Logging-Bundle zu tun. Was natürlich kein Mensch ernsthaft so sagen würde. Oder auch gerade, was ich noch meinte, die Dynamik an das Laufzeitverhalten der Anwendung. Es macht ja einen riesen Unterschied, ob irgendwo ein Funktionsaufruf ist und der wird einmal zur Initialisierung aufgerufen, sagen wir mal so einen Aufruf, wie, ich lade mir meine Konfiguration vom Konfigurations-Manager. So. Hat fast jeder irgendeine Dependency drauf, wird aber einmal geholt und das könnte auch genauso gut anders funktionieren. Und dann hat eben, was Alex ja auch grad meinte, oder halt irgend ne andere Funktion, die dann im Lebenszyklus dann 5.000 mal aufgerufen wird, weil das total eng zusammenhängt. Genau

und also der ganze Teil fehlt letztendlich komplett, wenn man nur diese Struktur anschaut. Und noch 'n ganz anderer, sehr wesentlicher Teil ist, RCE ist es ja auch eine verteilte Anwendung. Und das wird ja in sowas wie jetzt der IslandViz überhaupt nicht repräsentiert. Weil wenn man sich jetzt bei der IslandViz RCE schaut hat man keinerlei Anzeichen dafür, dass das in irgendeiner Weise eine verteilte Anwendung ist. Diese Information ist also überhaupt nicht vorhanden. Aber das ist eine ganz prägende Sache. Wenn jetzt jemand neu ins Team kommt, guckt sich RCE an und lässt dann komplett die ganze Verteilung heraus, dann wird er das System komplett falsch verstehen und auch Fehler einbauen.

I: Da ich jetzt immer wieder zur Historie will: ändert sich die Verteilung. Also RCE als verteiltes System ändert sich das, wo das hin verteilt wird? **B2:** Das ändert sich im Bezug worauf, über die Zeit oder?

I: Ja genau. Also das verteilt meinst du im Sinne von es läuft auf verschiedene Maschinen?

B2: Genau

I: Und ändert sich das, dass eine Komponente eine Funktionalität, in welcher Ebene wir auch immer reden, wird die einmal auf einem Rechner zugeteilt und bleibt dort immer auch oder können die auch verschoben werden?

B2: Ach so, ok, das ist es keine verteile Anwendung in dem Sinn, dass die Applikationen aufgeteilt wird und dann verteilt deployed wird, sondern es ist so: mehrere Instanzen der vollständigen Software werden auf verschiedene Systeme ausgerollt und die kommunizieren miteinander über ein Peer-to-Peer-Netzwerk.

I: Das heißt, es gibt die Inselwelt sozusagen mehrmals auf verschiedenen Rechnern.

B2: Ja

I: Ändert sich das, wie viele... Ja wahrscheinlich, dass immer mehr dazu kommen?

B2: Das entscheidet der Nutzer. Die Benutzer entscheiden ob sie ein RCE lokal nutzen oder auch nur ein Stück davon lokal.

I: Meine Frage vorhin mit der Historie von Software-Architektur: Wichtig oder nicht? Wurde über die Definition der Software-Architektur ein bisschen übergangen.

B2: Aber sonst ist die Frage halt nicht zu beantworten.

I: Ja klar, natürlich, aber ich wollte jetzt darauf zurückkommen. Deswegen nochmal die Frage, nachdem wir jetzt festgestellt haben, dass Software-Architektur nicht nur die Verteilung auf die unterschiedlichen Dateitypen im Endeffekt ist, sondern auch was kommuniziert, welche Komponente kommuniziert mit welcher wie stark: Schaut ihr euch diese

Historie, diese konzeptionelle Historie an? Ist die wichtig? Und in welchen Fällen ist sie wichtig?

B3: Also ich kann eigentlich sagen, dass ich im Moment immer noch genug beschäftigt bin, die aktuelle Architektur komplett zu verstehen. Und da halt auch gerade den Teil, den ich gerade für die aktuelle Aufgabe brauche. Und, also einmal habe ich mir tatsächlich noch nie überlegt mir die bisherige Architektur mal anzuschauen. Ich wüsste auch gar nicht, wie ich es tun sollte, ich wüsste auch nicht, was es mir bringen würde. Einfach auch weil, das mag ein bisschen fatalistisch klingen, aber die Architektur ist ja so wie sie im Moment ist, und Architektur-Entscheidungen werden ja auch häufiger bewusst getroffene, mehr oder weniger, jetzt im Vergleich zu Entscheidungen auf Code-Ebene. Auf Code-Ebene kann's auch wirklich sein, dass es irgendwelche Flüchtigkeitsfehler oder so was gibt, da ist es halt praktisch mit Hilfe der Historie rauszufinden, warum eine Änderung gemacht wurde. Bei den Architektur-Entscheidung glaube ich jetzt nicht, dass das da häufig passiert, dass die, in Anführungsstrichen, ausversehen getroffen werden.

B2: Also ich wüsste auch so gut wie keinen Fall, wo ich mal wirklich in der Historie schauen würde. Also ich wüsste nicht welche Fragen ich damit beantworten wollen würde. Entweder nur so aus historischem Interesse, so wirklich für Software-Archäologie, aber nicht um jetzt hier und jetzt ein konkretes Problem zu lösen. Da wüsste ich jetzt kein Beispiel. Oder halt, auch das könnte man sagen ist vielleicht bisschen fatalistisch, aber man ist meistens damit schon mehr als genug beschäftigt, die aktuelle Architektur zu verstehen, dass man nicht noch die weitere Zeitdimension mit drin haben will, weil das macht es nur komplizierter und bringt meistens eher wenig.

B4: Es gibt auch einfach sehr selten Änderungen, gut jetzt sind wir wieder bei der Frage der Definition, aber ich wüsste jetzt gar nicht wann sich die Architektur mal so wirklich stark geändert hätte.

B1: Es gab mal Pläne (. . .) Aber das zeigt ja schon, dass das nicht der Alltag ist.

B2: Und auch wenn, also es gibt schon mal Architekturänderungen, es natürlich immer die Frage wie grundlegend. Also richtig grundlegende Architektur von so einer großen Applikation schmeißt man nicht mal eben um. Da bräuchte man schon einen sehr, sehr triftigen Grund und auch die Menpower, das durchzusetzen. Haben wir beides nicht. Detailänderungen gibt es natürlich ständig. Aber auch da, dann hat man ja irgendwie jetzt einen Grund warum das alte nicht mehr gute ist. Und hat eine Idee, wie man's verbessern kann. Und dann geht man von A nach B und mehr braucht man nicht. Wenn ich jetzt überlege, wann ich wirklich mal historisch geschaut hätte, wie war das früher architektonisch. Also

ich wüsste jetzt ein einziges Beispiel. Das ist halt so wie wir jetzt bestimmte Plug-Ins sag ich jetzt mal registrieren, also für die Themen halt die Komponenten registriert werden. Da weiß ich halt, einfach weil ich dabei war: Okay da gab es mal Änderungen und da habe ich dann irgendwann mal reingeschaut, wirklich wie war das früher zuerst geschrieben worden? Und hab dann mal geschaut ok, wie ist es umgebaut worden. Aber das war auch reines historisches Interesse. Das hat mir jetzt beim Lösen des jetzigen Projekts praktisch nichts gebracht.

I: Und du wusstest dann praktisch ungefähr indem und dem Jahr haben wir das geändert. Das heißt ich gucke in einem von den Commits von davor wie es aussah. Oder wie muss man sich da denn Ablauf vorstellen.

B2: Ja das war mehr so: ich wusste, es war früher anders, bin dann relativ beliebig in nen Zeitpunkt damals gesprungen, wo ich wusste, das müsste hinkommen. Ich wusste auch noch wer dran gearbeitet hat. Und auch geschaut, ja da gab's dann auch Commits tatsächlich und hab dann einfach nur geschaut wie war der alte Stand. Also ich habe mir nicht wirklich die Historie angeschaut im Sinne von Veränderungen, sondern ich habe eigentlich nur einen alten Stand aus dem Repository rausgekramt.

I: Genau. Und den Zeitpunkt des Standes, den du dir anguckst, den wusstest du einfach aus Erfahrung?

B2: Habe ich einfach grob abgeschätzt, ob das so hinkommt. Also ich habe eigentlich mehr so einen Diff zum alten Softwarestand gemacht als wirklich die Historie angeschaut.

I: Aber man kann ja die Historie sowohl linear zurück gehen als auch gezielt irgendwo hinspringen. So wie es für mich klingt, springt ihr eher gezielt irgendwo hin, als dass ihr euch das chronologisch durchschaut.

B1: Also was man ja bei der Fragestellung auch beachten muss: Also, sich in der Historie Architektur anzuschauen ist halt nicht so simpel mal gerade zu machen wie halt irgendwie, keine Ahnung, in irgendwie Dateien letztendlich Änderungen zu verfolgen. Weil Architektur ist ja grundsätzlich erst einmal jetzt in unserem Repo nicht visualisiert. Das heißt man muss ja schon sozusagen die Art der Architektur und wie sie umgesetzt ist kennen letztendlich, man muss dann gezielt wissen in welchen Dateien spiegelt sich denn Architektur wider oder in welchen Strukturen letztendlich. Das sind ja nicht nur Dateien, das sind ja Strukturen. Und das ist ja jetzt nichts wo ich sage, oh, ich gucken wir mal grad an wie ist die Architektur da gewesen. Sondern, wenn ich zu so einem Zeitpunkt springe, dann kann ich halt nicht in eine Datei reingucken und sagen: oh hier, Change-Set Architektur so und so. Ich mein, du kannst mich korrigieren, aber das ist natürlich was, wo man also global

über die Codebasis gucken muss und sich mit der Architektur letztendlich grundlegend auskennen muss, um die Änderungen überhaupt zu erkennen. Nichtsdestotrotz, ich glaube nicht, dass man es nicht tut, weil es nicht die simple Lösung dafür gibt. Sondern ich glaube auch, dass das was ist, was nicht so hohe Relevanz hat letztendlich.

B2: Natürlich ein ganz generell Thema natürlich auch bei Architektur, Visualisierung oder was immer alles, ist ja auch, dass ganz viel von der Architektur auch gar nicht im Code drin ist. Architektur ist ja auch sehr oft, das kenne ich so unter dem Begriff Rationale-Management, das ganz oft die Entscheidung, warum ist das so gewesen halt auch wirklich Teil der Architektur ist und nicht nur die Frage, wie ist jetzt das End-Resultat oder die endgültige Entscheidung dann in den Code gegossen worden. Ein konkretes Beispiel dafür sind einfach Performance-Erwägungen. Wenn man genau weiß, man könnte diesen Code auch anders schreiben, aber dann wäre die Performance schlechter. Das sieht man jetzt dem Code, der da steht, aber praktisch nicht an. Das wäre jetzt sowas. Idealerweise steht das im Code-Kommentar dran. "Hier haben wir so geschrieben, weil aus den und den Gründen"Manchmal ist es natürlich auch so ein bisschen weiches Wissen einfach, was man dann hoffentlich dann so von Person zu Person weitergibt und versucht zu erhalten oder halt durch Zufall drauf stößt.

B1: Die Architektur kann auch Dokumentation sein. Im Wiki, oder, keine Ahnung. Das kann ja auch alles. Also wenn man Konzepte niederschreibt, kann auch alles dazu gehören.

I: Gibt es bei euch so Stellen, wo Architektur Entscheidungen dokumentiert werden?

B2: Nicht systematisch. Also ich ganz persönlich mach's so: ich versuch halt sehr viel Code-Kommentar dran zu schreiben, gerade auch solche Entscheidung. Aber die habe jetzt keine Vorgabe oder kein festes Schema im Team.

B4: Ich glaube, es kommt sehr auf die Größe der Entscheidung an, würde ich sagen, also kleinere Sachen, die stehen dann vielleicht im Code oder auf 'm Mantis-Issue, also wenn man jetzt einen Issue aufmacht und sagt, ok ich möchte jetzt hier diese Stelle mal refactorn, weil... Und dann kann man das da vielleicht nachschauen. Noch andere Sachen, die jetzt viel Konzeptänderungen enthalten, die stehen dann vielleicht im Wiki irgendwo dokumentiert. Aber das muss man dann auch eben erst mal wissen, dass man an der Stelle nachgucken kann.

B2: Vielleicht noch so 'n bisschen anekdotisch, so fürs Protokoll: Ich merke, dass teilweise wirklich, wenn ich arbeite und gerade solche Architekturentscheidungen, oder auch einfach nur Implementierungsentscheidungen manchmal treffe. Ich merke oft, dass ich den Impuls hätte es in die Commit-Kommentare zu schreiben: "Change so und soünd dann noch er-

klären will warum: Üm das zu erreichen oder weil das so und so sein muss". Lass ich aber doch meist raus, weil sonst die Commit-Kommentare extrem sperrig und unübersichtlich werden. Aber es gibt keinen klar definierten Ort, wo ich diese Information sonst hinpacke. Außer in die Commit-Kommentare.

I: Das heißt man kann das Ganze jetzt so ein bisschen zusammenfassen Zum einen die Historie der Architektur ist eher schwierig zum einen für euch herauszufinden zum anderen benötigt ihr sie aber auch gar nicht. Was ihr an der Historie benötigt sind zum einen die relativ aktuellen Änderungen, einfach um zu sehen, was passiert ist, was gerade ansteht. Und zum anderen aber um Fehler oder auftretende Probleme lösen zu können, um zu sehen wann ist das hinzugekommen was das Problem auslöst. Dann wäre ich so mit meinem Teil eins zum allgemeinen Software-Architektur-Historien-Teil durch. Dann würde ich jetzt mich ein bisschen mehr Richtung Visualisierung bewegen. Besteht Bedarf einer Pause? Braucht jemand Kaffee?

(Pause)

I: Jetzt geht's also ein bisschen mehr Richtung Visualisierung und dann auch ein bisschen mehr Richtung IslandViz. Wie gesagt, mir ist bewusst, dass es lange nicht alles abbildet was man eigentlich sehen wollte. Aber wir tun jetzt einfach mal, so als ob es das könnte was es wollte beziehungsweise, wir gucken einfach mal, was man machen kann momentan mit dem was momentan da ist. Kennen eigentlich alle die IslandViz? Ihr habt jetzt schon ganz viel erzählt, dass ihr im Code geguckt habt, verschiedene Sachen... Malt ihr euch dann UML-Diagramme nebenher, dass man guckt so das UML-Diagramm war früher so und ist jetzt so oder so. Gibt's da /

B1: Vereinzelt, also ganz selten.

B2: Also grundsätzlich ist es so, was ich gewöhnlich sehr oft male sind so die klassischen Kästchen und Linien Diagrammen und nach allem was man so hört ist das auch quasi der Industriestandart. Das machen so ziemlich alle Projekte. und diese ganzen UML-Diagramme-Typen höchstens ganz, ganz speziell. Also so 'n Sequenz-Diagramm vielleicht manchmal noch, aber wie gesagt, das ist auch schon fast Ende der Fahnenstange. Ok, und Zustandsdiagramme für State-Machines, das natürlich auch noch, das wär's aber auch schon so ziemlich, würde ich sagen.

B3: Ich muss ganz ehrlich sage, ich persönlich, wenn ich selbst, per Hand irgendwas male, dann sind's meistens eher so Sequenzdiagramme, weil Klassendiagramme könnte man sich am ehesten noch generieren lassen. Sequenzdiagramme sind da schon sehr viel schwieriger. Und für Klassen-Zusammenhänge nehme tatsächlich eher das, was Eclipse mir schon

bietet.

I: Und könnt ihr euch vorstellen, dass man ein altes Diagramm, in welcher Form auch immer, mit einem neuen vergleicht, um zu sehen, was sich geändert hat, oder nicht, oder ist das schon zu viel Bild?

B2: Also abstrakt vielleicht, aber ich habe in all den Jahren, wo ich Software-Entwickle, diesen Bedarf noch nie gehabt. Das wäre ja auch höchstens Softwarearchäologisch eben, ne so Ächwie war's denn früher mal.Äber nicht um eine konkrete Frage zu lösen.

B3: Ist eigentlich wieder das gleiche Problem wie vorher ich hätte keine Ahnung, welche Frage ich damit konkret beantworten sollte.

B4: Ich weiß auch nicht, ob man, wenn man jetzt wirklich irgendwie zwei Diagramme hat, ob man die Änderungen da wirklich so nachvollziehen kann. Es reicht ja schon, wenn man 'n paar Klassen umbenennt, dass man dann eigentlich die Zusammenhänge nicht mehr sieht und eigentlich von Hand gucken müsste so welche Klasse in dem neuen Diagramm entspricht denn welcher in dem alten. Das finde ich jetzt nicht so leicht anhand der Diagramme irgendwie nachzuvollziehen.

I: Wenn wir uns jetzt die IslandViz vorstellen, wie sie ist, oder nicht wie sie ist, sondern wie wir sie gerne hätten, die IslandViz kann jetzt das, was wir was wir uns vorstellen, was sie können soll und soll jetzt Historie zeigen. Ganz blöde Frage: aber wie würdet ihr euch das vorstellen? Was würde sie dann zum Beispiel können?

B1: Ich frag mich gerade, was die IslandViz für mich können soll. Überhaupt. Also ich weiß nicht, ob wir da alle den gleichen Stand haben. Aber, also bis jetzt ist das was für mich, auch jetzt nicht nur den aktuellen Stand, den ich sehe, auch alles was ich mir vielleicht da weiter drunter vorstellen könnte, ist das aus Entwicklersicht, um mich produktiver zu machen noch nicht das was ich mir vorstelle. Also insgesamt nicht.

I: Ok.

B1: Da kann ich vielleicht irgendwie schick irgendwelche Leute mit beeindrucken die vielleicht nicht wirklich konkret und ernsthaft am Code arbeiten, aber als Entwickler komme ich immer ganz schnell wieder auf eine Code-Ebene und da hab ich Entwicklungsumgebungen, die mir einfach Tools und Werkzeuge an die Hand geben, wo ich mich ganz schnell komplett in der kompletten Codebasis navigieren kann und, meiner Meinung nach, im Moment, zu mindestens, mit all dem, was ich kenne und mir vorstellen kann, einfach dass das mächtigere Werkzeug und bessere Werkzeug am Ende ist.

B5: Ja, selbst für Neueinsteiger in Software kann ich mir auch gar nicht vorstellen, dass

einem die IslandViz, selbst wenn sie jetzt noch optimiert wäre, da wirklich 'n großer Gewinn wäre.

B3: Sagen wir mal so, was ich mir vorstellen könnte, wäre für diesen Use-Case von auf den aktuellen Stand bringen, was in den letzten vier Wochen oder so passiert ist. Das man da vielleicht hingehen könnte und in dieser IslandViz zeigen könnte, ok auf den und den Inseln hier war letztens Aktivität oder so was. Oder die Inseln sind so ein bisschen gewachsen oder geschrumpft oder irgend sowas. Da wüsste ich jetzt ehrlich gesagt nicht, welchen Mehrwert mir diese Darstellung als Inseln bringt. Aber das ist ja eh jetzt kein Task zum richtig produktiv arbeiten. Das ist ja nicht, ich habe irgend ne Aufgabe, die ich erfüllen muss, sondern es ist eher die Sache, ich will ja einfach nur gucken, was ist in meinen drei bis vier Wochen Urlaub passiert. Und da denke ich, würde mir die Insel-Visualisierung auf jeden Fall nicht schaden.

B2: Die Frage da würde ich halt nur aufwerfen: Momentan sind die Inseln ja, entsprechen ja, was wir eben hatten, den Software-Strukturen. Wenn mir das die IslandViz zeigen will historisch: ok, hier an diesen Gebäuden blinken jetzt gerade Lichter, weil da hat sich was getan. Das bringt mir keinen Mehrwert, da guck ich lieber ins Subversion-History, ich meine, das ist die gleiche Information, aber übersichtlicher. Wenn schon so was wie eine Insel-Metapher, dann würde ich jetzt mal, so wünsch-dir-was-Modus, würde ich mir denken, dann sind die Inseln zum Beispiel, sagen wir mal thematische Cluster. Semantische Sachen, also im Sinne von das ist jetzt irgendwie dieser Themenkreis oder vielleicht auch dieses Feature, Branch wäre jetzt vielleicht schon zu technisch, aber sagen wir mal, das ist jetzt dieser Feature-Bereich oder dieser architektonische Bereich. Und da hat sich etwas getan. Das wäre kondensierte Information, die vielleicht interessant wäre.

I: Ok, das heißt, dass auch eine Funktionalität wieder auf mehrere Bundles wieder verteilt sein kann?

B2: Absolut. Also es ist sogar, glaub ich, also könnte ich jetzt nicht zu 100 Prozent sagen, aber ich glaube, das ist schon gar nicht selten. Weil man hat ja mindestens in jeden Fall, wenn da eine neue Funktion reinkommt, zumindest schon mal ein Backend-Teil, das da ist, einfach, und ne GUI-Entsprechung und oft auch sowas wie Konfigurationssachen, die natürlich da auch noch mit reinspielen. Und wenn das wirklich schonmal gruppiert werden könnte, das wäre schonmal sinnvoller.

I: Ok. Und dann könnte man aber auch wieder hervor, was hat sich in den letzten vier Wochen geändert.

B2: Genau, das kann man machen. Ganz ketzerisch natürlich, kann man sich jetzt fragen,

ob dann, also das ist ja erstmal unabhängig von der Frage ist es VR oder ist es nicht VR. Natürlich könnt man sich diese Art von Kondensation auch vorstellen, dann krieg ich einfach einen schönen Report einfach: Es gab die und die Änderungen in den und den Themenbereichen. Ob man das dann in VR macht oder nicht.

B1: Also ich bin noch so einem Stück bei dir, muss ich sagen. Ich glaube aber, man kommt letztendlich, um die Information zu erhalten, die man haben möchte, immer wieder entweder wirklich auf Text oder auf Code zurück. Also diese Gebäude, Inseln, Häfen bringen für mich persönlich keinen Mehrwert.

B2: Da denke ich auch ganz speziell an die Commit-Kommentare. Ich denke auch, ohne dass die da auftauchen, würde ich mich schon sehr eingeschränkt vorkommen. So wie coden mit dicken Winterhandschuhen. Und dann ist natürlich die Frage, ja wenn ich dann diesen Text dann irgendwie mit in die 3D-Welt einbauen, aber ist das dann ergonomischer als in perfekter Text-Auflösung auf dem 2D Bildschirm zu haben.

I: Vor allem natürlich auch ermüdend, weil an der IslandViz muss man stehen und sich umgucken, während man am Rechner sitzt und den Bildschirm genau in der Richtigen Höhe hat.

B2: Das ist jetzt nicht genau die Frage, die du gestellt hast mit der Historie oder so was in 3D. Aber mehr die Frage, wo könnt denn VR sinnvoll sein? Wo ich mir schon eher vorstellen könnte, wäre sowas wie gerade so auch das dynamische Verhalten von so einer Anwendung oder auch das Laufzeitverhalten. Entweder jetzt wirklich aktuell: ich guck wirklich auf ein Live-Systeme, oder ich habe eine Aufzeichnung von einem laufenden System. Wenn ich da z.B. mir anschaue könnte, ich habe also die ganze Applikation als, sagen wir mal Inseln, oder irgendeine andere Struktur. Und dann fährt die Applikation hoch, und dann sehe ich sozusagen blinkende Lichter, wo in der Applikation ist gerade wieviel Aktivität. Wie, was man vielleicht so kennt, so ein funktionaler Hirnscan, welche Hirnzentren sind jetzt gerade aktiv. Weil da kann ich mir vorstellen, da könnte man vielleicht bei so einer VR-Sache wirklich so ein bisschen intuitiv so Muster erkennen. Dass man sieht, ok beim Hochfahren: erst überall ist alles schön verteilt und dann plötzlich ist da dieser kleine Knubbel, der glüht und da bleibt dann der ganze Applikations-Stand erstmal zehn Sekunden hängen und da glüht alles. Daraus könnte ich intuitiv was schließen, was in anderen Medien glaub ich nicht so leicht erkenne.

I: Oder dann natürlich auch, wenn ich die und die Nutzer-Eingabe mache, dann fängt es woanders wieder an.

B2: Ja genau sowas. Oder, wenn ich jetzt so einen Workflow starte, auch da zu gucken, wie

diffundiert die Aktivität durch das ganze System, vielleicht irgendwann sogar mal verteilt, wenn ich dann wirklich sagen könnte, ok, ich will zwei Applikationen, die ich parallel sehe, man startet den einen Workflow, der den anderen mit einbezieht, dann auch sehe, wie da die Aktivität zwischen denen so hin und her fließt oder Ping-Pong spielt oder so.

B1: Oder auch so Datenflüsse.

B2: Genau, Datenflüsse, Aufrufe, Rechenzeit-Verbrauch vielleicht sogar RAM-Verbrauch, das wären so Sachen, da könnte ich mir VR schon vorstellen. Vielleicht auch eine kleine Zusatz-Anekdote: Ich hatte auch mal auf einer Konferenz auch mal mit einem sehr bekannten Entwickler aus dem OSGi-Umfeld, halt so eine Koryphäe aus dem OSGi-Ökosystem auch mal gesprochen, hab ihn genau das gefragt. Ich habe ihm das IslandViz-Paper auch gezeigt und hab ihn auch gefragt: Wo würdest du dir das vorstellen? Und seine Antwort ging auch in eine ganz ähnliche Richtung. Er kommt natürlich mehr aus dem Business Umfeld. Aber er meinte er könne sich gerade so was vorstellen so eine Visualisierung, wenn zum Beispiel die ein komplexes OSGi-System haben im Produktionsbetrieb und irgendwas läuft nicht. Wenn er da z.B. so visuell sehen könnte wo z.B. sind irgendwelche Services z.B. gerade nicht gestartet. Weil man das natürlich visuell sehr gut erfassen kann und vielleicht auch sogar, wenn man das normal laufende System halt kennt, einen visuellen Eindruck hat, wie sieht das System normal aus. Und dann sind Menschen ja ziemlich gut darin, diese Unterschiede zu finden und dann sieht: ok, jetzt das System, die Ecke sieht irgendwie anders aus. Was man vielleicht in so einem technischen Diff nicht so schnell erkennen würde.

B1: Also viel mehr Fokus auf Laufzeitverhalten als auf Entwicklungszyklus.

B3: Da wäre es tatsächlich zu fragen mit der Historie, ob dann interessant, wenn jetzt, sagen wir mal, RCE beim Start-Up neuerdings hängt. Da wäre dann tatsächlich auch der Unterschied interessant, ne alte und ne neue Version zu haben, die parallel starten. Und dann würde ich als Mensch wahrscheinlich auch relativ schnell erkennen, wenn ich jetzt wirklich so ein flackerndes Lichtmuster hab, oh, das sieht aber ab der Stelle hier, oder ab der Stelle fängt es an sich da und da an sich zu konzentrieren. Und wenn man weiß, dass das jetzt schon vorher war, dann ist der Knubbel jetzt wahrscheinlich nicht das Problem.

I: B4, hast du Wünsche, wir sind im Wünsch-dir-was-Modus.

B4: Also wie gesagt, das mit dem Laufzeitverhalten finde ich eine ganz gute Idee. Das glaube ich auch, dass das hilfreich sein könnte. Also ich finde das ja wie es jetzt ist mit den Inseln vielleicht mal so auch als sich veränderndes System zu visualisieren, das könnte auch sinnvoll sein, allerdings vielleicht nicht für uns als Entwickler, um damit irgendwas

zu machen, sondern wirklich um eine schöne Darstellung für Außenstehende zu haben. Also, ich denke die ganze Zeit an dieses Video was wir immer so bei den Konferenzen laufen haben, mit der Commit-Historie. Das ist ja auch nichts, was man als Entwickler nutzt und dann guckt wo ist denn da was commited worden. Aber es ist halt einfach eine schöne Darstellung wo man sieht, wie jetzt sich die Codebasis so entwickelt. Also so auf der Ebene könnte ich mir das mit der IslandViz auch gut vorstellen. Da irgendwie über die Zeit neue Inseln auftauchen oder neue Verbindungen entstehen, sich was aufteilt.

I: Hattet ihr schon mal einen Fall, dass ihr jemand im außenstehendes, zeigen müsst, dass ihr auch arbeitet, in Anführungsstrichen, dass ihr das irgendwie demonstrieren wolltet oder dass derjenige etwas sehen wollte?

B1: Naja, im Endeffekt ist das ja schon wieder ein Stück weit mehr Alltag. Allerdings, ich sag jetzt mal, wir haben irgendwelche Ziele, die wir erreichen wollen mit unserer Entwicklung und letztendlich zählt dann da das Ergebnis. Und das Ergebnis, das ist dann entweder in Form von irgendeinem Feature, was nachher tatsächlich in der Software ist, oder irgendwie einer Verbesserung, irgendwas halt, wirklich was man in der Software sieht letztendlich. Das ist ja das, was man dann sozusagen zeigt bzw. ausliefert letztendlich. Und wenn wir von Projekten reden, dann schreiben wir halt auch in den Projekt-Report rein, was wir gemacht haben. Das ist aber was, was man uns dann auch glaubt, wenn wir das da reinschreiben. Dass müssen wir dann nicht unbedingt dann nochmal untermauern.

B4: (scherzhaft) Wir schreiben einfach in den Bericht: Kommt und guckt euch das in der IslandViz an.

B1: Also ich sag jetzt mal so, ich habe diese IslandViz jetzt auch schon auf der einen oder anderen Konferenz präsentiert und das ist das, was ich eben auch eingangs so meinte, aus Entwickler-Sicht sehe ich da noch nicht so den Mehrwert. Ich kann mir durchaus vorstellen, dass man das mit so einer Visualisierung Mehrwert erzeugen kann, aber halt auf ganz anderen Ebenen. Also irgendjemandem, der sozusagen, dem das technische Verständnis oder der Entwickler-Background oder IT-Background fehlt, der irgendwie eine Software sieht, was in seinen Augen erstmal irgendeine grafische Oberfläche ist, dem irgendwie zu erklären, was da alles so hinter steckt vielleicht auf einem wirklich technischen Level, was nicht viel Wissen erfordert. Für sowas kann ich mir das durchaus gut vorstellen. Hier kursierte auch schonmal, irgendwie das Beispiel, vielleicht das irgendwie einem Manager zu zeigen. Um halt zu sagen genau, das haben wir getan. Joa, wenn ein Manager sich davon beeindrucken lässt, dann würde ich das Management in Frage stellen. Also nur mal so. Das weiß ich nicht, kann aber sein. Aber so aus Entwickler-Sicht, würde ich sogar, zumindest

im Moment wahrscheinlich, eher vermeiden so etwas zu zeigen, um zu beweisen: wir haben auch wirklich was getan. Da würde ich andere Wege wählen.

B2: Wobei das ja auch das generelle Problem, was man mit Metriken einfach hat. Letztendlich kann die Visualisierung ja auch nur irgend ne Form von Metriken anzeigen. Und dass Metriken nicht unbedingt mit der Produktivität der Arbeit oder mit der Nützlichkeit der Arbeit korrelieren ist ja auch hinlänglich bekannt. Z.B. in diesem, was wir eben hatten, in dieser Baum-Animation von der Historie von RCE, die da schon als Video existiert. Ich weiß nicht, ob die noch irgendwann mal gefiltert worden ist, oder was raus ist, aber in irgendeiner Version war z.B. mal, dann lief diese Animation ab, und dann gab's irgendwann ein total bombastisch aussehendes Ereignis, wo der ganze Graph plötzlich einmal aufblitzte. Und dann haben, auf der FroScon oder so, Leute, meine ich auch nachgefragt. Ja, was war dann passiert, da hatte ein Student ausversehen einen Trunk gelöscht. Und der wurde im nächsten Commit entsprechend dann wieder zurückkopiert. Das sah natürlich in der Darstellung halt aus wie ein bombastisches Ereignis.

B4: Ja, sowas mit Jahreszahlen ändern, ist da auch schön. Alle Klassen angefasst.

B2: Ja, in jeder Klasse Copy-Right so und so. Und rums: Alle Dateien im Projekt angefasst.

B2: Ich hatte eben schon überlegt, wo es darum ging irgendjemandem, der nicht so technisch unterwegs ist, was man dem sinnvoll damit visualisieren könnte. Man könnte ja vielleicht mal zeigen: Der Benutzer sieht die grafische Oberfläche und dem Benutzer jetzt mal klar zu machen, ok du siehst hier oben diese grafische Oberfläche, aber ist sozusagen nur die Spitze des Eisbergs, da drunter kommt noch ein riesiger Brocken Back-End, der macht richtig viel Arbeit. Klingt erst mal völlig nach grundsätzlich einer ganz guten Idee, hat aber das Problem: GUI-Code ist relativ geschwätzig, sehr, sehr sperrig. Das heißt, egal nach welcher Metrik man da geht, nach Code-Zeilen oder wie auch immer, wird die GUI da durchaus groß aussehen. Das ist dann wirklich, gerade diesen Lerneffekt gerade nicht kriegt. Also ich will nicht sagen, dass das nicht geht. Man müsste schon sehr genau überlegen was da die Metrik ist. Diese Aktivität, was wir vorhin hatten, dieses Laufzeitverhalten, das könnt ich mir tatsächlich spannend vorstellen. Wenn der Benutzer vielleicht sogar, gut müsste man jetzt praktisch gucken, wie das gehen würde, aber im Sinne von der Benutzer sieht wirklich: er klickt in der GUI auf diesen Button und was passiert dann dahinter ein System. Vielleicht auch gar nicht unbedingt, dass es nützlich ist, vielleicht ist es auch einfach nur cool, interessant aussieht, was, so ein Gefühl dafür zu kriegen was passiert.

B5: Aber aus Demonstrationszwecken, um nochmal auf die IslandViz auch zurückzukommen, kann ich mir das schon auch so gut vorstellen, auch da die Historie dann abzubilden. Weil ich glaub, das ist jetzt auch nicht der aktuelle Status, dass man dann halt sieht wie Inseln wachsen und vielleicht wieder zerstört werden oder wie auch immer. Dass es ein bisschen, ja, noch weiter animiert wird und nicht nur diesen statischen, jetzt sieht man ja den einen aktuellen Stand, welcher auch immer das ist, das wäre schon nett, glaube ich, aber wie gesagt nur zu Demonstrationszwecken.

B2: Das wäre natürlich, sowas eben gerade dieses Inseln entstehen und vielleicht verändern die auch ihren Umfang (. . .) Das wäre natürlich auch wieder was interessantes, wenn man das auch auf diese thematischen Gruppen runterbrechen könnte. Also wie sind Funktionsbereiche im Gesamtsystem vielleicht gewachsen über die Zeit. Das könnte ich mir auch interessant vorstellen.

B1: Dann ist nur die Frage, wie man sowas halt automatisieren soll. Weil die ja erstmal nicht so vorliegen.

B2: Aber wir sind hier ja in der Forschung.

B1: Und bei Wünsch-dir-was.

I: Ja, da würde dann, dass man erst mal die Daten irgendwie anders extrahieren muss.

B1: Ja vor allen Dingen, ist diese Information in der Form gar nicht unbedingt direkt vorhanden. Also man müsste ja diese Funktionsbereiche definieren tatsächlich. Und ich wüsste jetzt nicht, ob wir das einfach so könnte.

B5: Und die Definitionen sind ja vielleicht auch nicht statisch. Also die könnten sich ja auch im Laufe der Zeit verändern. Dann kommen vielleicht, wachsen Bereiche zusammen, oder so.

B2: Man müsste anfangs sicherlich erstmal wirklich jemand haben, der hingeht und das quasi verschlagwortet. Kann man für so ein Forschungsprojekt natürlich machen, aber, genau, wie B1 sagt, für ein unbekanntes Projekt wäre es natürlich schön, wenn es auch irgendwie automatisch ginge.

B4: Man kann natürlich versuchen, so Bereiche irgendwie dann wirklich erst mal ganz einfach über ne Graph-Analyse zu finden. Aber ich glaube, da kommt dann auch viel raus, was man eigentlich nicht haben will. Es gibt ja viele Bundles, die quasi mit allen Zusammenhängen: Logging oder GUI oder so. Deshalb glaub ich, würde das nicht so komplett automatisiert funktionieren, dass man jetzt sagen kann, so das gehört jetzt zu einem Bereich alles.

B1: Vor allen Dingen, also ich sag jetzt mal, ganz blöd die Frage, aber, was ist ein Feature. Das muss man ja erst mal definieren für so eine Software.

B2: Gut, ich mein, das ist jetzt nicht die Frage. Aber, ich glaub man könnt aus der Historie natürlich versuchen zu schauen, woran wurde parallel gearbeitet, vielleicht auch als Informationsquelle, vielleicht aus der Richtung irgendwie.

I: jetzt gehen wir noch einen Schritt weiter weg von dem Entwickler zu: Wir wollen es jemandem zeigen, der eigentlich keine Ahnung hat. Und ich habe schon ein paar Dummy-Ideen gemacht, die ich gern einfach irgendjemanden zeigen würde, was man jeweils schöner findet. Nur so, um mal eine Meinung zu haben. Und zwar geht es mir vor allem um die Bedienbarkeit. Weil ihr kennt die IslandViz. Momentan, man hat dieses Tablet auf dem der Name der Klasse steht wo, man noch die Sachen ein und ausblenden kann und man kann rein und raus zoomen. Wenn ich jetzt da Historie zu baue, dann muss natürlich auch zusätzliche Nutzer-Interaktion passieren, und zwar zum einen wie navigiere ich durch die Historie. (Eine Kollegin), hat das ja in 2D gemacht, das kriegen wir dann am Montag präsentiert, die navigiert durch den Zeitstrahl entweder indem sie einfach Schritte vor und zurück geht oder indem sie so etwas wie einen Zeitraffer ablaufen lässt. Was ich jetzt ganz häufig hier gehört habe, was ich auch auf jeden Fall in irgendeiner Form einbauen will ist, dass man sich vorher ein Commit aussucht, wo man dann hin springt. Jetzt aber zu der Vor- und Zurück- Navigation: Was ich mir zum einen vorstellen könnte, ist dass man einfach das Tablet erweitert und sagt ich habe einen Step-Vorwärts, Step-Rückwärts Buttons oder auch Zeitraffer-Vorwärts, Zeitraffer-Rückwärts, Zeitraffer-Anhalten Button auf dem Tablet zum Navigieren. Was ich mir auch vorstellen könnte, ist dass man das auf dem Tisch direkt als Buttons macht. Wie gesagt, das ist jetzt alles noch nicht hübsch. Das kann man programmieren, dass die Buttons immer direkt vor einem auf dem Tisch erscheinen egal wo am Tisch ich stehe, dass vor mir diese Buttons sind und ich darüber dann vorwärts, rückwärts laufen kann. Das ist so der eine Aspekt, der andere ist dann, ich habe ja gesagt, ich werde auf jeden Fall in irgendeiner Form die Commit Historie haben, da habe ich momentan die Überlegung, dass man das auch wieder auf dem Tablet eventuell macht, dass man an dem Datum, wobei das Datum habe ich festgestellt gar nicht so interessant ist. Ich habe dann noch, dass man das so aufklappen kann und dann z.B. die Commit-Message lesen kann. Oder aber, dass man auch zu dem Commit, wo ich gerade bin die Information, das ist sozusagen so eine Infowand, ne Pinnwand mit in das Ganze integriere. Wo ich da eben entweder die Information, welchen Commit schaue ich mir gerade an, oder aber, wo bin ich eigentlich in der Projekt-Zeit gerade, sich das so durchschauen kann. Ich würde gern einfach mal Meinungen dazu hörten oder vielleicht,

ob ihr das benutzbar findet?

B2: Ganz spontaner Eindruck, also erste das hier, was wir gerade sehen, Folie 15, das, wie du schon sagst, das Datum selbst ist völlig irrelevant, allermeistens. Und dann das, was auf Folie 16 ist, das mit dem Ausklappen, das fände ich von der Bedienung ziemlich grausig, da tipp ich mich tot und sehe nichts. Das, was die nächsten beiden Folien, 17 und 18, zeigen, das finde ich erst mal gut, wenn ich in einem Zeitstand bin, dass ich da was als Referenz habe, sehr nützlich und diese Timeline auf 18 finde ich grundsätzlich auch nützlich. Allerdings muss man aber natürlich auch fairer Weise einschränken, das ist jetzt hier natürlich ein sehr entspanntes Beispiel. Wie nützlich das ist, wenn da mal 80 Commits sind, das ist dann wieder ne andere Frage. Und ich habe hier natürlich nicht die Meta-Information zu den Commits. Aber einfach so, ich mein, das könnt man sich einfach wie so ein Histogramm vielleicht vorstellen, anstatt dass es einfach so am Meeresstrand steht, einfach so ein bisschen so 'n Aktivitätsgrafen über die Zeit, das ist sicherlich ein nettes Feature. Ob es die ganze Zeit da sein muss ist eine andere Frage. Und zu den Sachen, die am Anfang waren, also die wären mir jetzt persönlich sehr steuerflächenlastig, wo ich mich frage, warum? Wir sind hier in VR, wir wollen jetzt nicht den Desktop nachbauen, weil wenn ich nen Desktop haben will, benutze ich den Desktop. Also meine allererste Idee, wie ich in so ner Zeitlinie navigieren würde in VR wäre sowas wie: ich hab den Controller in der Hand und ich drehe den. Sowas wie Ich definier eine Anfangs- und eine Endzeit und dann ist zum Beispiel ganz den Controller ganz nach links gedreht der Anfangszeitpunkt und - da muss man ergonomisch gucken, wie weit nach rechts sinnvoll ist- und dann ganz nach rechts, wie so ein Steuerrad, wo ich meine Hand drehe und damit scroll ich durch die Zeitlinie. Weil, ich will jetzt nicht mit VR einen Scrollbalken, den ich anklicken muss und den ich nach links und rechts ziehe, nachbauen, das macht für mich keinen Sinn. Oder halt eine andere Geste, was weiß ich, Handfläche hoch runter heben oder nach links rechts schieben.

B5: Ich habe nur mal eine Frage dazu, zu der Folie 17. Die Commit-Message würde sich dann beziehen jetzt auf exakt wieviel Commits von dem Tag?

I: Ne, auf den Commit von 15:02 Uhr.

B5: Was schaue ich mir dann zu dem Zeitpunkt an?

I: Also ich würde, es ist jetzt natürlich, weil das Screenshots aus der inaktiven IslandViz sind mit Bildern rein montiert sind, auf dem Tisch wäre dann die IslandViz zu dem Zeitpunkt, also die Insel mit ihren Verbindungen zu diesem Zeitpunkt und auf der Wand steht eben wann ist das und was wurde dazu geschrieben.

B3: Zu der Folie 15, wo das Tablet mit den Überschriften. Ich denke um einen einzelnen Commit auszuwählen, ist die Ansicht an sich gar nicht mal so schlecht. Nur würde ich wahrscheinlich Datum und Uhrzeit gegen Autor und Commit-Message-Titel austauschen. Jede Commit-Message hat ja, die erste Zeile wird ja meistens als Titel benutzt, und darunter kann man dann Fließtext schreiben. Wenn man das irgendwie reinfrickeln könnte, dass das beides in eine Zeile passt, mit Autorennamen und dem Titel könnte ich jetzt sehr viel besser identifizieren, was das zu welchem Commit ist. Und wann der gemacht wurde ist mir eigentlich vollkommen egal, solange die in der richtigen Reihenfolge sind.

I: Das hab ich jetzt auch aus dem Gespräch auf jeden Fall raus gehört, das Datum bringt erstmal gar nichts.

B3: Und was auch praktisch wäre: wenn man das dann nach Branches filtern könnte. Dass man, wie B2 das am Anfang auch meinte, sagen kann ich möchte jetzt gerne nur die Commits haben, die auf den Branch zutreffen.

B1: Also insgesamt Filtermöglichkeiten vielleicht auch noch Autoren fände ich auch sinnvoll und wo ich auch voll bei B2 bin, ist, dass man, wenn man sich in so einer Welt bewegt mit Steuerflächen hab ich so digitale Steuerung, ein Step, Step, Step. Die Steps bei so Commits, die können beliebigen Umfang haben. Das kann sein, dass ich mit einem Commit einen Typo gefixt habe, dann hab ich einen Buchstaben geändert, oder ich hab irgendwie ne komplett neue Klasse hinzugefügt und schon viele Lines of Code. Und wenn ich halt so eine Art Steuerung habe, die halt proportionalen funktioniert letztendlich, dann kann ich natürlich mit irgendeiner Bewegung auch sehr schnell durch so eine Historie bewegen. Das heißt, ich kann vielleicht, so zu sagen, Gas geben und schneller in den Zeitstrahl langgehen und wenn ich merke, ich komme in den richtigen Bereich, dann ist das Ganze langsamer wieder bewegen. Das würde ich auf jeden Fall als viel sinnvoller erachten.

B2: Kurz noch ein kleines Detail zu der Sache mit glaub 17. Hier fände ich jetzt von der Usability noch so ein bisschen so die Sache. Das ist ja nicht ganz match, weil ich sehe jetzt ein Ist-Zustand, weil es ist ja quasi ein Repository-Zeitpunkt und ich sehe das Commit-Info, aber das Commit-Info ist ja eigentlich das Diff von dem Jetzt-Zustand auf den Schritt davor. Also diese Anzeige würde für mich jetzt vor allen Dingen Sinn machen, wenn ich jetzt hier in der Visualisierung irgendwie das Ding zum Vor-Commit sehen würde.

B5: Da würde ich z.B. zumindest die Inseln, oder was auch immer da angefasst worden ist, könnten irgendwie markiert werden.

I: Hat noch jemand entweder zudem, also zu den Beispielen Anmerkungen oder zu dem, was wir sonst besprochen haben, oder nochmal ganz eine andere Idee zum Thema Historie.

Dann wäre ich nämlich mit meinem Teil am Ende. (. . .) Ich bedanke mich nochmal ganz herzlich dafür, dass ihr teilgenommen habt.

Anhang C. Evaluation: Einführungstext

Zur visuellen Unterstützung der Abschnitte des Einführungstexts wurden Bilder verwendet, die hier jeweils am Ende des Abschnitts referenziert sind.

Grundlegende Beschreibung der dargestellten Software-Architektur

Das dargestellte Software-Projekt gliedert sich in sogenannte Bundles, die auch als in sich abgeschlossene Module betrachtet werden können. Die Bundles enthalten jeweils Packages als weitere Gliederungsebene. Die Packages wiederum enthalten die einzelnen Klassen und Interfaces der Software.
Ein Bundle kann auf Funktionalitäten aus einem anderen Bundle zugreifen, in dem es daraus Packages importiert.
(Abbildung C.1)

Inselmetapher

In der Anwendung wird diese Gliederung des Software-Projekts durch eine Inselmetapher dargestellt. Dabei wird jedes Bundle durch eine Insel (a) repräsentiert. Auf der Insel befinden sich verschiedenfarbige Regionen (b), die jeweils die einzelnen Packages des Bundles darstellen. Auf den Regionen befinden sich Gebäude (c), die die Klassen und Interfaces darstellen.
Die Inseln sind auf einem Meer verteilt.
Die Importbeziehungen zwischen den Bundles werden durch Pfeile dargestellt, die von den Häfen (d) der Inseln ausgehen.
(Abbildung C.2)

Navigation in der Anwendung

Mithilfe der Controller können Sie innerhalb der Inselwelt navigieren. Die Controller haben sowohl ein Touch-Pad (Frontansicht, großes rundes Feld) als auch einen Trigger (Seitansicht oben), mit denen Sie eingaben machen können.
(Abbildung C.3)

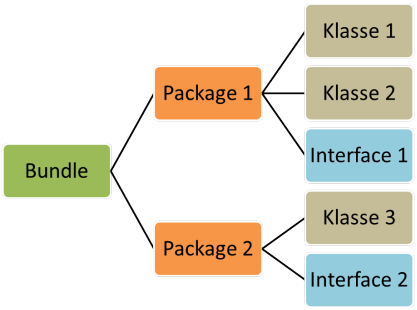


Abbildung C.1

Sie können die **Ansicht verschieben**, indem Sie den **Trigger eines Controllers** gedrückt halten und den Controller dicht über dem Meer **hin und her** bewegen.
(Abbildung C.4)

Sie können die **Ansicht vergrößern und verkleinern** (bzw. rein- und raus-zoomen), indem sie die **Trigger beider Controller** gedrückt halten und die Controller über dem Meer **voneinander weg** (vergrößern) oder **aufeinander zu** (verkleinern) bewegen. (Ähnlich wie mit zwei Fingern auf einem Handy-Display gezooomt wird)
(Abbildung C.5)

Sie können die **Ansicht drehen**, indem Sie die **Trigger beider Controller** gedrückt halten und die Controller **kreisförmig bewegen**.
(Abbildung C.6)

Um die Inseln näher zu erkunden, haben Sie einen Laserpointer. Dieser wird aktiviert, indem Sie ihren Daumen auf das Touch-Pad legen. Wenn Sie auf das Touchpad drücken wird das mit dem Laserpointer ausgewählte Objekt markiert.
(Abbildung C.7)

Sie bekommen gleich ausreichend Zeit sich in der Anwendung mit diesen Navigationsmöglichkeiten vertraut zu machen und können dabei gerne Fragen stellen.

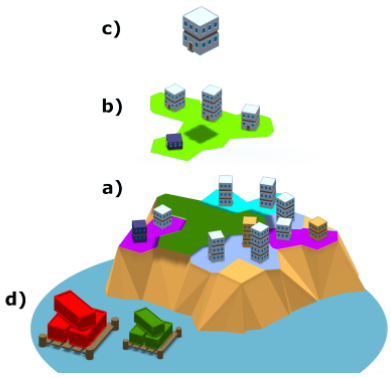


Abbildung C.2. (Schreiber, Nafeie, Baranowski, Seipel & Misiak, 2019)

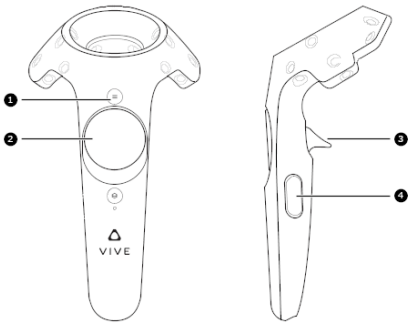


Abbildung C.3. (DLR-SC, 2020)

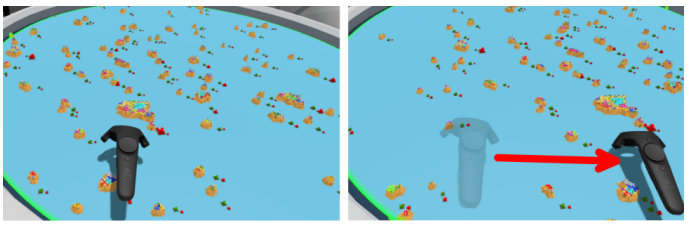


Abbildung C.4. (Schreiber, Nafeie, Baranowski, Seipel & Misiak, 2019)

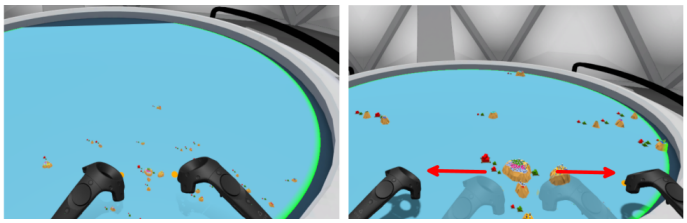


Abbildung C.5. (Schreiber, Nafeie, Baranowski, Seipel & Misiak, 2019)

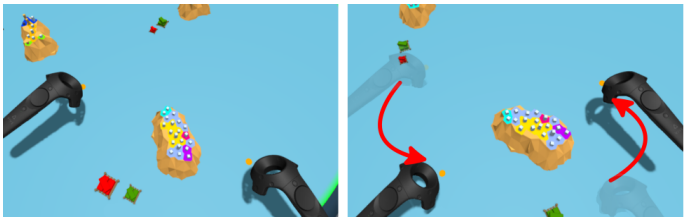


Abbildung C.6. (Schreiber, Nafeie, Baranowski, Seipel & Misiak, 2019)

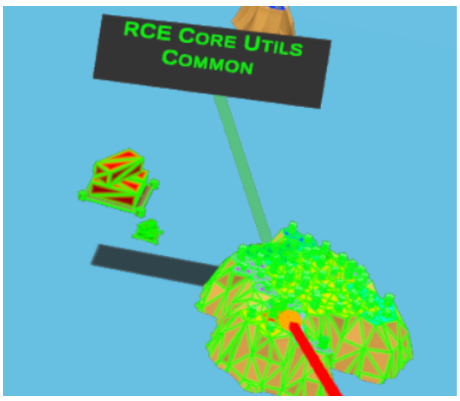


Abbildung C.7. (Schreiber, Nafeie, Baranowski, Seipel & Misiak, 2019)

Software-Historie

Die Anwendung ermöglicht es Ihnen auch, den Zustand des Software-Projekt zu verschiedenen Zeitpunkten zu betrachten (verschiedene Commits im Versions-Verwaltungssystem). Dazu können Sie mittels der Pfeiltasten in der Anwendung

>| einen Zeitschritt vorwärts gehen

|< einen Zeitschritt rückwärts gehen

Use Case

Stellen Sie sich vor, Sie sind Projektleiter eines Software-Projekts. Das heißt, Sie programmieren nicht selbst und sind auch nicht mit allen Details der Software-Architektur vertraut. Sie bestimmen lediglich, welche Funktionalitäten noch hinzugefügt werden müssen und teilen die Aufgaben unter Ihren Mitarbeitern auf.
Nun möchten Sie im Nachhinein nachvollziehen, wie groß der Umfang der Änderungen war und welche Teile der Software geändert wurde.

Anhang D. Evaluation: Visualisierungsvarianten

D.1 Visualisierung der Systemebene

Das gesamte Software-System in zwei aufeinanderfolgenden Commits

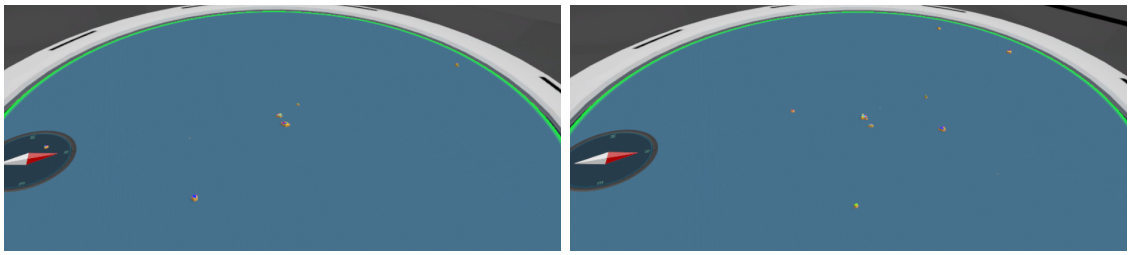


Abbildung D.1. System a): Die Position der Inseln wird neu berechnet, ohne die vorherigen Positionen zu berücksichtigen.

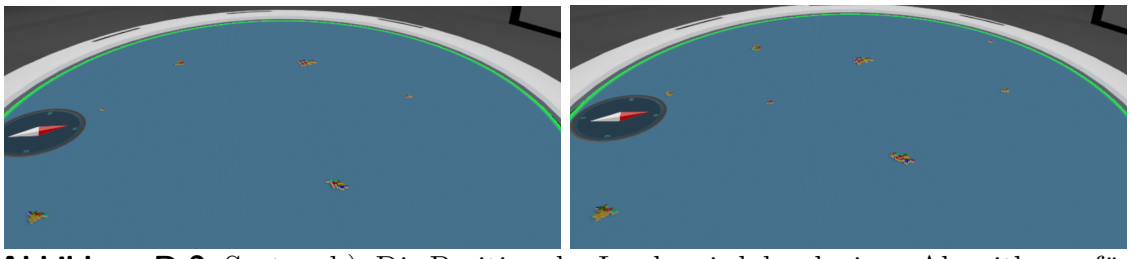


Abbildung D.2. System b): Die Position der Inseln wird durch einen Algorithmus für dynamische Graphen bestimmt, so dass die Inseln an ähnlichen Positionen bleiben.

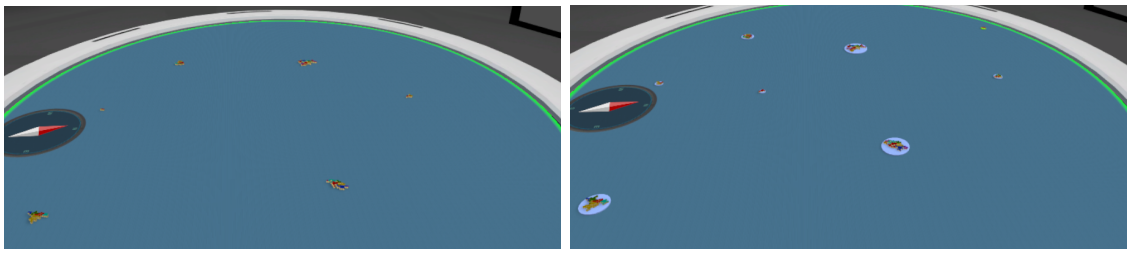


Abbildung D.3. System c): Die Position der Inseln wird wie in System b) berechnet, Änderungen an den Inseln und neue Inseln werden farblich hervorgehoben.

D.2 Visualisierung eines Bundles

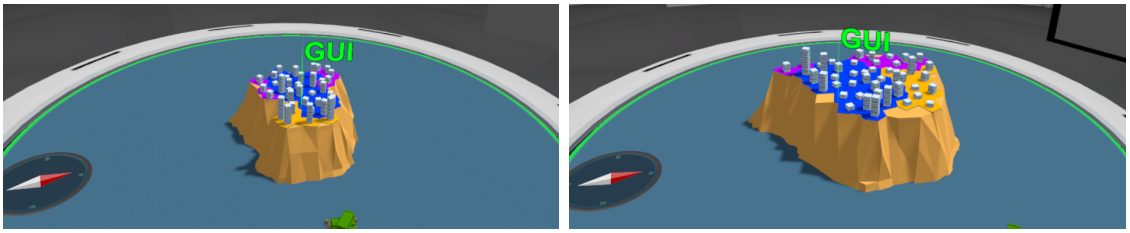


Abbildung D.4. Visualisierung eines Bundles in zwei aufeinander folgenden Commits in System a): Das Layout der Regionen, die Farbe der Regionen und die Position der Gebäude wird neu berechnet, ohne das vorherige Layout zu berücksichtigen.

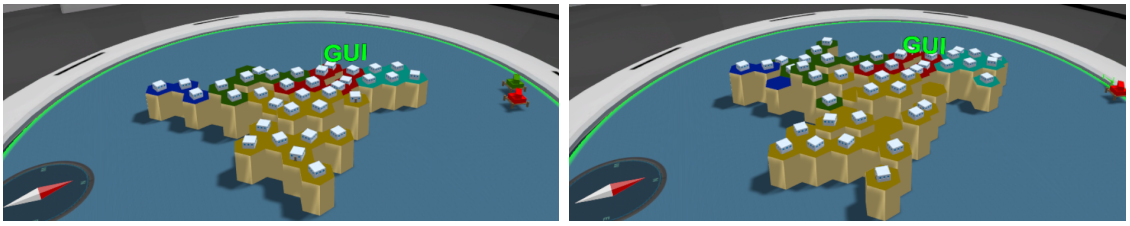


Abbildung D.5. Visualisierung eines Bundles in zwei aufeinander folgenden Commits in System b): Das Layout der bereits vorhandenen Regionen und Gebäude wird beibehalten. Neu Regionen und Gebäude werden außen an die Insel angehängt. Die Topologie der Insel spiegelt die Lebenszeit der CompilationUnits wieder. Die Hexagon-Zellen gelöschter Gebäude bleiben leer.

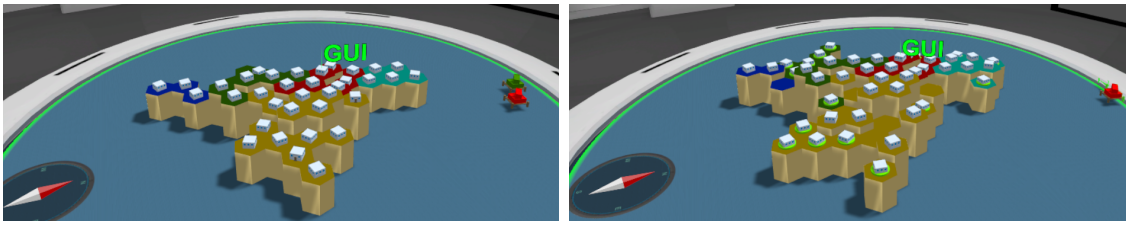


Abbildung D.6. Visualisierung eines Bundles in zwei aufeinander folgenden Commits in System c): Das Layout der Insel wird wie in System b) berechnet, neue Gebäude und Gebäude mit geänderter Höhe werden farblich hervorgehoben.

Anhang E. Evaluation: Qualitative Ergebnisse

Im folgenden werden die Freitextantworten und Aussagen der Probanden vollständig, in ungeordneter Reihenfolge aufgeführt.

E.1 Freitextantworten

E.1.1 Was hat Ihnen beim Übergang zwischen den Zeitpunkten gefallen?

System a)

- gute und interessante Idee
- einfache Bedienung zwar noch sehr eingeschränkt, aber es hat auf alle Fälle Potential
- Dass man die verschiedenen Zustände visuell vergleichen konnte
- Eindeutiger Unterschied zwischen den beiden Zeitpunkten
- In derselben Optik und Systematik gehalten.

System b)

- Leichte Bedienung
- Der Bildausschnitt ist ungefähr gleich geblieben, sodass man gut vergleichen konnte
- Keine abrupte Veränderung der Zustände
- Bei Aufgabe 2: Die Insel bleibt im Fokus, und man muss sie nicht erneut auswählen.
- Bei Aufgabe 1: Visualisiert sieht man hinzugekommene & verschwundene Inseln im Direktvergleich
- Man sieht sofort die Unterschiede
- Dass man an den Inseln schnell sieht, ob Packages geändert wurden

System c)

- Freundliche Benutzeroberfläche, einfach zu bedienen
- Durch die Farbmarkierungen konnte man direkt die Veränderungen sehen (außer bei neuen/gelöschten Packages)
- Farbliche Markierung der Inseln und Gebäude bei Veränderungen
- Die grünen Ringe haben Veränderungen sehr gut dargestellt. Im Prinzip das, was ich mich bei der vorherigen Anwendung (ohne Ringe) gewünscht hätte: Eine Darstellung oder Historie der einzelnen Veränderungen in den Inseln.
- Man konnte sehen, was geändert wurde
- Dass die Veränderungen zur zuvor angesehenen Version sofort ersichtlich sind

E.1.2 Was hat Ihnen beim Übergang zwischen zwei Zeitpunkten nicht gefallen?

System a)

- wenige oder geringe Informationen, schwieriger Vergleich der Zustände der einzelnen Inseln
- Inseln haben ihre Platzierung /Location gewechselt, oder scheinbar gewechselt
- Ich hätte es besser gefunden, wenn ein Bundle an das man herangezoomt hat nach einem Zeitsprung auch wieder herangezoomt dargestellt worden wäre. So musste man jedes Mal erneut heranzoomen
- Ladezeit, grauer Bildschirm
- Zu extreme visuelle Änderung, durch die einfarbige Fläche habe ich mir schwer getan mir weiterhin das zu merken, was ich mir merken wollte.
- Schlecht, da Ladevorgang, zwei gefühlt getrennte Systeme. Änderungen nicht im System wahrnehmbar, muss aus dem Gedächtnis verglichen werden. Damit fehleranfällig, bei Kontrolle muss Ansicht wiedereingestellt werden. Inseln gefühlt nicht am gleichen Fleck.

System b)

- Wenn das Informationsfenster ein wenig höher angezeigt wird, ist vielleicht angenehmer den Überblick über das Meer zu behalten.

- Das Bild hat ziemlich hin und her gewackelt und man musste jedes Mal warten, selbst wenn man sofort wieder zum vorherigen Zeitpunkt zurück wollte.
- Keine Markierung oder Hervorhebung der Veränderungen
- Kurze Wartezeit beim Wechsel zwischen Commits. Einige Inselnamen wurden nach dem Wechsel nicht angezeigt, mussten im Menü erst erneut angezeigt werden lassen.
- In Aufgabe 2: Die Insel im Fokus ist aus unerklärlichen Gründen immer etwas "davon geschwommen"
- Dass sich die neu geladene Insel bewegt hat bzw. etwas von mir weggeschwommen ist.
- Dass sich die Karte verschiebt, auch wenn man auf eine Insel zentriert ist

System c)

- wieder das Wackeln bzw Verschieben des Bildes
- Gelöschte Packages, Bundles und Dateien verschwinden einfach. Es fällt schwer, mehrere Löschungen auf einmal nachzuvollziehen, vor allem, wenn die Package-Ebene auf der Insel im Gegensatz zu den anderen Package-Ebenen abgesenkt ist, sodass niedrige Gebäude hinter anderen verschwinden.
- Die Namen einzelner Inseln wurden nicht angezeigt und mussten im Menü erst wieder aktiviert werden.
- Man hat nicht auf den ersten Blick gesehen, welche Pakete dazugekommen sind bzw. sich in der Größe verändert haben
- Dass im neuen Zeitpunkt nicht mehr vorhandene Dinge nicht mehr angezeigt werden

E.1.3 Was hat Ihnen an der Darstellung der Inselwelt gefallen?

System a)

- übersichtlich, einfach nachzuvollziehen, alles relativ offensichtlich aufbereitet, gut veranschaulicht
- einfache Visualisierung mit Verbindungen etc.
- Man kann dadurch die Größe der einzelnen Bundles und ihre Beziehungen untereinander einschätzen
- Ansprechendes Design, farbliche Kodierung, Übersichtlichkeit

- Je Zeitpunkt war die Anfangsdarstellung identisch - es ging nicht zu der Position zurück, die zuletzt in diesem Zeitpunkt ausgewählt wurde
- Interessante Darstellung der komplexen Unterstrukturen. Durch Höhe der Gebäude werden Codefülle sichtbar, ohne dass es ablenkt oder stört. Durch Connections / Interfaces sind Querverbindungen sichtbar.

System b)

- Bunte Farben, übersichtliche Gruppierung
- Farben sind sehr gut voneinander unterscheidbar, Package-Größe ist durch die Wabenform sehr gut erkenntlich
- Sehr anschaulich und einfach verständlich
- Sechseckige Darstellung + Farbliche Unterschiede unterstützen die Anschaulichkeit. Höhe der Häuser stellt Klassen-/Interfacegröße dar, dadurch Bezug zur Realität und intuitives Verständnis der Welt.
- Es hat eine intuitive Art der Nutzung. Die Farben machen es sehr übersichtlich
- Innovativer, überblicklicher Eindruck eines Projekts

System c)

- wie vorher, gute Erkennung der Package-Größe durch Wabendarstellung
- Hervorhebung der Veränderungen sehr gut deutlich durch auffällige/leuchtende Farben
- Die Ringe haben die Darstellung deutlich verbessert und das Augenmerk in gewollte Richtungen gelenkt.
- sehr übersichtlich
- Farbige Kennzeichnung, welche Bundles sich geändert haben

E.1.4 Was hat Ihnen an der Darstellung der Inselwelt nicht gefallen?

System a)

- Wegen hoher Dichte der Gebäude Auswahl manchmal nicht "nebenbei"möglich

- Die Häuser waren relativ eng beieinander, so dass es schwer war sie mit dem Laser zu markieren. Außerdem hatten nebeneinander liegende Bereiche z.T. die gleiche Farbe
- Die Positionen der Inseln waren nicht über die Zeitpunkte hinweg an derselben Stelle.
- Allgemein nur Überblick möglich, keine Querverbindungen darstellbar

System b)

- Die großen Häuser verdecken die Sicht auf die kleineren
- Die einzelnen Bundles werden schnell unübersichtlich durch die Anzahl der Gebäude (eventuell größere Abstände?).
- Die Inseln sind teils sehr weit auseinander, manche sind sehr klein sodass man sie erst bemerkt, wenn man stark vergrößert. Beim Vergrößern der Inselwelt sind durch den Abstand außen gelegene Inseln dann jedoch nicht mehr auf dem Schirm.
- Inseln recht klein und weit auseinander, viel Scrolling nötig

System c)

- wie vorher, man kann die kleineren Häuser hinter den größeren schlecht erkennen
- Außerdem habe ich bei einer sehr kleinen Insel in der Gesamtansicht nicht erkennen können, dass sie grün hinterlegt war"
- Bei sehr kleinen Inseln waren grüne Ringe teils sehr unscheinbar, bei Heranzoomen waren dann andere Inseln wieder nicht im Bild.
- Farbliche Markierung ziemlich klein und ganz ausgezoomt schwer zu sehen

E.1.5 Was hat Ihnen an der Anwendung insgesamt gefallen?

System a)

- gute Visualisierung eines Zeitpunktes, gute Visualisierung der Beziehungen der einzelnen Klassen, Packages und Bundles, intuitive Nutzung, Daten waren gut und relativ übersichtlich aufbereitet
- witzige Idee für das Management von Softwareprojekten
- ganz gut, ist sehr übersichtlich

- Interaktion und spielerische Veranschaulichung
- Einfaches Handling, einfache Navigation, ansprechendes Design (Farben, Formen,...)
- Interessante Darstellung, gute Bedienbarkeit in einem Commit Zeitpunkt. Alles für einen Überblick Nötige ist zugänglich.

System b)

- praktisch und leicht zu bedienen
- die Steuerung über die Controller ist angenehm und intuitiv
- Sehr einfach und intuitiv zu bedienen. Man kann sich einfach eine schnelle Übersicht schaffen.
- Bezug zur Realität macht die Interaktion und das Verständnis sehr intuitiv, man lernt schnell mit der Steuerung der Anwendung umzugehen. Die Tafel zur linken war nützlich (auch wenn ich sie zunächst vergessen hatte - Aus den Augen, aus dem Sinn).
- Es wirkt ein bisschen wie Arbeit mit gamification auf mich. Es macht Spaß mit dem Programm umzugehen
- Innovativ, guter Überblick

System c)

- Praktisches Nutzen, Übersichtlichkeit
- die Steuerung über die Controller ist angenehm und intuitiv
- Einfache Verständlichkeit der Bedienung und allgemein gute Übersicht durch die farblichen Hervorhebungen möglich.
- Verschafft guten Überblick über Veränderungen des Systems, im Vergleich zu Anwendung 1, bei der man dies händisch nachschauen musste.
- intuitive Nutzung, sehr übersichtlich
- Guter Überblick über das Projekt

E.1.6 Was hat Ihnen an der Anwendung insgesamt nicht gefallen?

System a)

- Zu wenig Infos über Änderungen zwischen 2 Commits

- Kein Speichern von zuletzt besuchter Insel beim Zurückkehren in einen Commit
- gestört hat mich am meisten das Zoomen nach den Zeitsprüngen und, dass nebeneinander liegende Bereiche die gleiche Farbe hatten
- Farbfläche beim Laden war wie ein Papier vor dem Gesicht, bei dem man trotzdem am Rand noch den vorherigen Zeitpunkt erkennen konnte.
- Umsetzung der Commit Zeitpunkte als jeweils eigene Inselwelt.
- Keine Speicherung der Ansichten vor der Umschaltung in einen neuen Commit Zeitpunkt.

System b)

- Die Navigation mit den Controllern ein wenig optimieren
- Man muss sich relativ viel merken bzw. häufig hin und her springen, um sicherzugehen, dass man alle Veränderungen wahrgenommen hat
- Veränderungen von einem Commit zum nächsten nicht hervorgehoben
- Teils war das Menü der linken Hand störend im Weg. Ich wollte z.B. eine Insel anvisieren, öffnete jedoch nur mein Offhand-Menü.
- Die Steuerung war manchmal nicht ganz so, wie ich es erwartet hätte (man muss näher ans Wasser, oder klickt versehentlich etwas Falsches an...). Ist aber nur ein sehr kleiner Punkt.
- Für produktiven Einsatz fehlen noch ein paar Features, wie Versionsunterschiede

System c)

- diesmal ist mir die Benutzung teilweise schwerer gefallen, beim Zoomen/Drehen ist die Insel einmal komplett aus dem Bild verschwunden
- Bei nahem Zoom verschwindet manchmal die Insel, die man im Moment betrachtet aus dem Bild (Tisch) durch die Veränderung der Abstände.
- Kleine Inseln sind unscheinbar und die Ringe teils nicht gut zu erkennen, wenn man in der zentrierten Gesamtansicht der Inselwelt ist. Die unterschiedlichen Terrain-Höhen einzelner Inseln haben mich etwas verwirrt, da ich den Grund hierfür nicht erkannt habe.
- Laserpointer macht oft unerwünschte Dinge

E.1.7 Welche zusätzlichen Informationen oder Hilfen hätten Sie sich beim Lösen der Aufgabe gewünscht?

System a)

- Anzeige z.B. auf Tafel über Änderungen an der selektierten Insel, Package
- auf dem "Screen" wären Informationen zu den einzelnen Packages mit Namen etc. gut gewesen und ein Symbol, ob das Package neu ist
- Dass man zwei Bundles aus verschiedenen Commits nebeneinander / gleichzeitig ansehen kann und keine Zeitsprünge machen muss
- Angabe der Namen der Packages auf den Inseln und der Interfaces und Klassen in der Package-Information.
- Transparenz zwischen Commit Zeitpunkten, Farbige Markierungen von Zuwachs, transparente Inhalte kennzeichnen Löschungen etc.

System b)

- keine
- Eine Anzeige der gelöschten bzw. neu erstellten Elemente z.B. durch farbliche Markierung
- Farbliche Hervorhebung der Veränderungen.
- In der Tafel links eine Auflistung von gelöschten, hinzugefügten und veränderten Daten.
- Eine Anzeige auf der Tafel, welche Bausteine sich verändert haben (gelöscht, hinzugefügt). evtl. zusätzliche Menüpunkte auf der Tafel, um sich die Historie zwischen den Commits deutlicher anzeigen zu lassen.
- Anzeigen der Versionsunterschiede

System c)

- keine
- Auch bei den Packages eine farbliche Markierung der neuen Inselregionen
- Hervorhebung der gelöschten Elemente. Eventuell zum Meer senkrecht stehende Linien in der der Veränderung entsprechenden Farbe, die von einem veränderten

Gebäude aus nach oben gehen. Sonst übersieht man sehr schnell die Veränderung von kleinen Gebäuden.

- Die Tafel links könnte zudem dennoch eine schriftliche Historie á la Github geben.
- Wenn in der aktuell angezeigten Version ein Bundle oder ein Package nicht mehr vorhanden ist, wäre es einfacher zu erkennen, wenn dieser Bestandteil noch halbdurchsichtig o.ä. angezeigt würde

E.1.8 Was möchten Sie noch zu der Anwendung sagen?

System a)

- sehr guter Einsatz von VR zum Überblick behalten bei großen Projekten, gute Visualisierung.
- Könnte Potential haben
- Wirkt professionell erstellt.
- Interessant, für einen Überblick gut, aber für Veränderungen mehr Transparenz zwischen den Commit Zeitpunkten nötig.

System b)

- cool
- Insgesamt eine spaßige und übersichtliche Anwendung, die Projektleitern das Controlling erleichtern könnte. Auf Dauer kann ich mir vorstellen, dass die Anzahl der Inseln und Insel-Sektoren unübersichtlicher wird.
- Zudem strengt übermäßig langes Bedienen der Anwendung körperlich an.

System c)

- alles super
- Einfach und intuitiv, Veränderungen können schnell identifiziert werden, und das ist für Projektleiter immerhin das Wichtigste: Was hat sich getan seit dem letzten Mal?

E.2 Aussagen während der Studie

E.2.1 Aufgaben auf Systemebene

System a)

- Muss ich mir jetzt alle merken?
- Inselstruktur sieht anders aus, also von den Bundles
- Authentication war glaub ich beim anderen da drüben irgendwo.

System b)

- Gibt's da auch irgendwo ne Übersicht, was sich verändert hat.
- Das muss man sich alles merken?
- Ich bin ein bisschen verwirrt, weil diese Insel keinen Namen hat.
- Mein Gedächtnis ist für solche Aufgaben zu schlecht.
- Die Transformations-Message überdeckt die Namensschilder
- Wäre schön, auf Liste aufzuzählen, was geändert wurde.

System c)

- Man könnte das, was gelöscht wurde, mit ein einem roten Kreis markieren, oder so.

E.2.2 Bundleebene

System a)

- Oh, der merkt sich das nicht, wo ich war.
- Also bei internal ist was dazu gekommen, das sind zwei Klassen und beim anderen vier.
- Das ist ja auch fies, dass die die gleiche Farbe haben.
- Sonst sind überall gleich viele Klassen (Nachdem bei großen Packages die Größe aufgerufen und verglichen wurde)
- Ach so, die Klassennamen kann man sich nicht auf einmal anzeigen lassen.

- Ich hätte mir jetzt die Klassen Packageweise angeschaut und verlichen, weiß aber nicht, ob ich mir die alle merken kann.
- Jetzt noch die Klassen: das ist ein bisschen wie die Nadel im Heuhaufen.
- Ich habe angefangen zu zählen, wie viele Klassen in einem Package sind.
- Haben die Farben eine Bedeutung?

System b)

- Die Tafel updatet sich beim Commit-Wechsel erst, wenn man das Objekt nochmal anklickt.
- Mich irritiert, dass da ein Sechseck leer ist.
- Warum schwimmt die Insel von mir weg?
- Warum fährt die Insel weg?
- Ich wünsche mir, dass auf der Tafel auch die Veränderungen angezeigt wird.
- Ich denke mir halt auch, wie lange es dauert, das in Git nachzuschauen, also da ist das schon besser.
- Das sind doch so viele. Ich gehe das jetzt Package-Weise durch.

System c)

- Ist das normal, dass die Insel so weg schwimmt?
- Die Klassen sind gelöscht aber das Package „Deleted Package“ ist ja noch da.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht.

Die Arbeit hat in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Würzburg, 9. November 2020

Elke Franziska Heidmann